DAG Scheduling and Analysis on Multi-core Systems by Modelling Parallelism and Dependency

Shuai Zhao, Xiaotian Dai, Iain Bate
Department of Computer Science, University of York, UK
{shuai.zhao, xiaotian.dai, iain.bate}@york.ac.uk

***Abstract*—With ever more complex functionalities being implemented in emerging real-time applications, multi-core systems are demanded for high performance, with directed acyclic graphs (DAG) being used to model functional dependencies. For a single DAG task, our previous work presented a concurrent provider and consumer (CPC) model that captures the node-level dependency and parallelism, which are the two key factors of a DAG. Based on the CPC, scheduling and analysis methods were constructed to reduce makespan and tighten the analytical bound of the task. However, the CPC-based methods cannot support multi-DAGs as the interference between DAGs (i.e., inter-task interference) is not taken into account. To address this limitation, this paper proposes a novel multi-DAG scheduling approach which specifies the number of cores a DAG can utilise so that it does not incur the inter-task interference. This is achieved by modelling and understanding the workload distribution of the DAG and the system. By avoiding the inter-task interference, the constructed schedule provides full compatibility for the CPC-based methods to be applied on each DAG and reduces the pessimism of the existing analysis. Experimental results show that the proposed multi-DAG method achieves an improvement up to 80% in schedulability against the original work that it extends, and outperforms the existing multi-DAG methods by up to 60% for tightening the interference.**

I. Introduction

Driven by the demands of high performance and complex functionalities, multi-core systems with complex computing models are increasingly being deployed in real-time applications. The Directed Acyclic Graphs (DAGs) are often used to model functional dependencies between the parallel computation units in such systems [2]. Figure 1 provides an example DAG which contains eight nodes with a set of edges. A node indicates a computation unit that must be executed sequentially and a directed edge describes the execution dependency of two nodes (e.g., node v_5 and v_7). Nodes with no dependency can be executed in parallel, e.g., node v_2, v_3 and v_4. Many existing works use DAGs to model the system [3], [4], [5], [6], [7], [8], [9]. For example, Verucchi et al. [3] models a complete automotive task chain from perception to control as a DAG task.

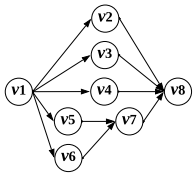


Figure 1: An example DAG.

Focusing on a single recurrent DAG task, Zhao et al. [10] reduced the makespan (i.e., the time interval between the start and finish of the DAG execution) and provided a tight yet safe bound on the makespan, compared to the state-of-art method in He et al. [1]. To achieve this, a model named the *Concurrent Producer and Consumer* (CPC) was developed in Zhao[10] to provide detailed knowledge of the node-level parallelism and inter-node dependency of a DAG task, which are the essence of the DAG topology. This paper was the first work that fully exploited the node-level parallelism and inter-node dependency of a DAG task to facilitate both schedule and analysis.

However, despite the reduced makespan and tighter analysis for a single DAG task, the methods proposed in Zhao[10] can only support executing one DAG at a time. This is because the limitation that the CPC-based methods did not take into account the interference between DAG tasks (i.e., inter-task interference). For systems with multiple recurrent DAG tasks [1], [11], [12], methods in Zhao[10] will lose their benefits as DAGs must be executed one by one in a sequential fashion even if certain cores are idle during the execution of a DAG, which results in a low schedulability. This limitation imposes a significant barrier for the methods in Zhao [10] to be applied on multi-DAG systems.

The principal contribution of this paper is a novel scheduling approach for multiple sporadic DAGs which (i) is the first multi-DAG method compatible with the CPC-based analysis constructed in Zhao[10] and (ii) significantly reduces the pessimism of the analysis. To achieve this, the following new contributions are made in Section VII. To achieve this, the following new contributions are made in Section VII:

- A *Parallelism-aware Workload Distribution Model* (P-WDM) that describes the workload distribution of a DAG task when executing on any given number of cores (Section VII-B).
- A P-WDM accumulation method that adds two P-WDMs under a given number of cores. This enables the understanding of the workload distribution of the system when multiple DAGs are running in parallel (Section VII-C).
- An offline core assignment method that utilises the P-WDM to determine the number of cores that a DAG can execute on so that it (i) does not incur interference from other DAGs and (ii) can meet its deadline. This method provides full compatibility for the CPC-based methods in Zhao[10] on a multi-DAG system and minimises the pessimism in the analysis (Section VII-D).

For each DAG in a hyper-period, the constructed schedule first specifies the number of cores it can utilise based on the P-WDM model of the task and the current system. This avoid the inter-task interference between DAGs during execution. Then, given the specified number of cores, the methods in Zhao[10] are applied to schedule and analyse nodes in a DAG without incurring the pessimism due to the inter-task interference.

With the constructed multi-DAG method, we address the application barrier and significantly expand the capability of methods construed in Zhao [10] on multi-DAG systems. Experimental results show that the proposed multi-DAG method achieves an improvement up to 80% in system schedulability over the non-work-conserving method in original paper that it extends, Zhao[10], and outperforms existing multi-DAG methods (e.g., He [1] by up to 60%) for tightening the inter-task interference.

Organisation: Section II presents the task and system model. Section III describes the state-of-the-art approaches in DAG scheduling and analysis. Section IV presents the CPC model that captures the

two key factors (dependency and parallelism) of the DAG. Based on the CPC, Section V presents the new response time analysis and Section VI describes the proposed schedule for a single DAG task. Then, for multi-DAG systems, Section VII identifies and addresses the key challenges of applying the proposed analysis, and presents a novel scheduling method to improve overall system schedulability. Finally, the evaluation is reported in Section VIII and Section IX draws the conclusion.

II. Task and scheduling Model

A. Task model of a single recurrent DAG

A DAG task τ_x is defined by $\{T_x, D_x, \mathcal{G}_x = (V_x, E_x)\}$, with T_x denoting its minimum inter-arrival time, D_x gives a constrained relative deadline, i.e., $D_x \leq T_x$, and $\mathcal{G}_x$ is a graph defining the set of activities forming the task. The graph is defined as $\mathcal{G}_x = (V_x, E_x)$ where V_x denotes the set of nodes and $E_x \subseteq (V_x \times V_x)$ gives the set of directed edges connecting any two nodes. Each node $v_{x,j} \in V_x$ represents a task that must be executed sequentially and is characterised by its Worst-Case Execution Time (WCET), $C_{x,j}$. For simplicity, the subscript of the DAG task (i.e., x for τ_x) is omitted when we consider a single DAG task.

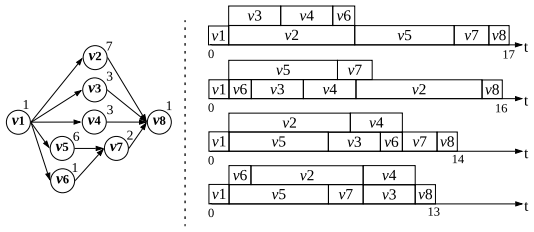
For any two nodes v_j and v_k connected by a directed edge ($(v_j, v_k) \in E$), v_k can start execution only if v_j has finished its execution. That is, v_j is a *predecessor* of v_k, whereas v_k is a *successor* of v_j. A node v_j has at least one predecessor $pre(v_j)$ and at least one successor $suc(v_j)$, formally defined as $pre(v_j) = \{v_k \in V \mid (v_k, v_j) \in E\}$ and $suc(v_j) = \{v_k \in V \mid (v_j, v_k) \in E\}$, respectively. Nodes that are either *directly* or *transitively* predecessors and successors of a node v_j are termed as its ancestors $anc(v_j)$ and descendants $des(v_j)$ respectively. A node v_j with $pred(v_j) = \varnothing$ or $succ(v_j) = \varnothing$ is referred to as the *source* v_{src} or *sink* v_{sink} respectively. As with He[1] and Fonseca[13], we assume each DAG has one source and one sink node. Nodes that can execute concurrently with v_j are given by $\mathcal{C}(v_j) = \{v_k | v_k \notin (anc(v_j) \cup des(v_j)), \forall v_k \in V\}$.

A DAG task has the following fundamental features. First, a path $\lambda_a = \{v_s, \cdots, v_e\}$ is a node sequence in V and follows $(v_k, v_{k+1}) \in E, \forall v_k \in \lambda_a \backslash v_e$. The set of paths in V is defined as Λ_V. A *local path* is a sub-path within the task and as such does not feature both the source v_{src} and the sink v_{sink}. A *complete path* features both. Function $len(\lambda_a) = \sum_{\forall v_k \in \lambda_a} C_k$ gives the length of λ_a. Second, the longest complete path is referred to as the *critical path* λ^*, and its length is denoted by L, where $L = \max\{len(\lambda_a), \forall \lambda_a \in \Lambda_V\}$. Nodes in λ^* are referred to as the *critical nodes*. Other nodes are referred to as *non-critical nodes*, denoted as $V^{\neg} = V \backslash \lambda^*$. Finally, the workload W is the sum of a task's WCETs, i.e., $W = \sum_{\forall v_k \in V} C_k$. The workload of all non-critical nodes is referred to as the *non-critical workload*.

Figure 2(a) shows an example DAG task with eight nodes (i.e., $V = \{v_1, v_2, ..., v_8\}$). The number at the top right of each node gives its WCET, e.g., $C_2 = 7$. Based on the above terminologies, for node v_7 we have $pre(v_7) = \{v_5, v_6\}$, $anc(v_7) = \{v_1, v_5, v_6\}$, $suc(v_7) = des(v_7) = \{v_8\}$ and $\mathcal{C}(v_7) = \{v_2, v_3, v_4\}$. For the DAG, we have $L = 10$, $W = 24$, with $\lambda^* = \{v_1, v_5, v_7, v_8\}$, $v_{src} = v_1$ and $v_{sink} = v_8$.

B. Task model of multiple recurrent DAGs

Following the single-DAG task model, a multi-DAG system contains n sporadic DAG tasks $\Gamma = \{\tau_1, ..., \tau_n\}$, in which each task τ_x is assigned with a unique priority P_x. A task τ_x can give raise to a set of jobs in one hyper-period. For the jth (starting from one) job of task τ_x, denoted as $J_{x,j}$, it is released at $r_{x,j} = T_x \times (j-1)$ and has a deadline of $d_{x,j} = D_x \times j$. As with the system model in He[1] and Fonseca[13], the DAG tasks in this work are independent from each other. That is, we assume DAGs do not share any resource and there exists no dependency between any two DAGs.



(a) Example DAG (b) Execution scenarios

Figure 2: Makespan of a DAG task with different execution scenarios.

C. System and scheduling model

In this work we consider a homogeneous system with m cores and a non-preemptive fixed-priority scheme for both DAG- and node-level scheduling [14]. That is, a late-arriving high priority DAG can be blocked by low priority DAGs that are running. The same rule also applies at the node level in one DAG.

The schedule for a multi-DAG system follows the principle of *highest priority DAG first* and then within a task, it follows the *highest priority node first*, for all DAGs tasks and nodes that are ready to release. That is, task priority is used to select the next task to execute in the ready queue, whereas node priority gives the exact execution order of nodes in the scheduled DAG. The priorities at DAG-level are assigned by the Deadline Monotonic Priority Ordering (DMPO) [15] and nodes in each DAG are ordered by Algorithm 2 presented in Section VI.

III. Related Work

A. Work-conserving schedule and analysis

The majority of the existing work on scheduling DAG tasks assumes a *work-conserving* scheduler [16]. A scheduling algorithm is said to be work-conserving if it never idles a processor when there exists pending workload. A generic bound that captures the worst-case response time of tasks scheduled globally with any work-conserving method is provided in Graham[17]. This analysis is later formalised in Melani[16] and Fonseca[13] for DAG tasks. The analysis of a single DAG task is given in Equation 1. Notation R_x denotes the response time of τ_x, m denotes the number of cores.

$$R_x = L_x + \left\lceil \frac{1}{m}(W_x - L_x) \right\rceil \tag{1}$$

In this analysis, the worst-case response time of a DAG task τ_x is upper bounded by the length of the critical path and the intra-task interference imposed by the non-critical nodes of τ_x itself. However, this analysis assumes the critical path can be delayed by all the concurrent nodes, which is pessimistic for scheduling methods with an explicit execution order known a priori [16], [1].

Figure 2(b) provides possible execution scenarios of the example DAG in a dual-core system. With nodes scheduled randomly, a total 240 different execution scenarios are possible, with a makespan ranging from 13 to 17. The analysis described above provides a safe bound with $R = L + \frac{1}{m}(W - L) = 10 + \frac{1}{2}(24 - 10) = 17$. However, there are scheduling orders with a makespan much lower than 17. Based on the work-conserving schedule and the classic analysis, we propose new analysis and scheduling approach to tighten the

analytical bounds of DAGs and to reduce the run-time makespan, respectively.

B. Scheduling a single DAG task

For homogeneous multiprocessors with a global scheme, existing scheduling (and their analysing) methods aim at reducing the makespan and tightening the worst-case analytical bound. They can be classified as either slice-based [18], [19] or node-based [1], [20]. The slice-based schedule enforces node-level preemption and divides each node into a number of small computation units (e.g., units with a WCET of one in Chang[18]). By doing so, the slice-based methods can improve node-level parallelism but to achieve an improvement the number of preemptions and migrations need to be controlled.

The node-based methods provide a more generic solution by producing an explicit node execution order, based on heuristics derived from either the *spatial* (e.g., number of successors of a node [21] and topological order of nodes [1]) or the *temporal* (execution time of nodes [20], [3], [22]) characteristics of the DAG. Below we describe two most recent node-based methods.

In Chen et al. [20], an non-preemptive scheduling method is proposed for a single periodic DAG, which always executes the ready node with the longest WCET to improve parallelism. Chen[20] prevents anomalies from occurring when nodes are executing less than their WCETs, which can lead to an execution order different from the schedule. This is achieved by guaranteeing nodes are executed in the same order as the offline simulation. However, without considering inter-node dependencies, this schedule cannot minimise the delay on the completion of DAG. For the example in Figure 2, this method leads to the scenario with a makespan of 14, in which the non-critical node v_6 delays the DAG completion due to a late start.

In He et al. [1], a new response time analysis is presented, which dominates the traditional bound in Graham[17] and Melani[16] when an explicit node execution order is known a priori. That is, a node v_j can only incur a delay from the concurrent nodes that are scheduled prior to v_j. Then, a scheduling method is proposed that always executes: i) the critical path first; and ii) the immediate interference nodes first (nodes that can cause the most immediate delay on the currently-examined path). The novelty in He[1] is considering both topology and path length in a DAG, and provides the state-of-the-art analysis against which our approach is compared. However, the method in He[1] schedules concurrent nodes based on the length of their longest complete path (a path from the source to the sink node), i.e., nodes in the longest complete path first. This heuristic is not dependency-aware, which reduces the level of parallelism that can be exploited, and hence, lengthen the finish time of a DAG task.

C. Scheduling multi-DAG systems

The above works focus on scheduling and analysing a single DAG task, but can be applied to multi-DAG systems with a global work-conserving scheme which dispatches ready nodes to the first available core. In addition, there exists a large body of work which has a particular focus on allocating DAGs to cores. They can be classified into two major categories: *federated* and *partitioned*.

The *federated* scheme considers DAG tasks as either heavy or light, where each heavy task is assigned with a number of dedicated cores and the light ones share the remaining cores. Li et al. [23] presents a hard real-time federated scheme for DAGs with arbitrary deadlines and achieves a capacity augmentation bound of 2. Baruah [24] presents a federated schedule for new DAG model (i.e., conditional DAGs) with both constrained and arbitrary deadlines. Ueter et al. [25] provides an reservation-based federated scheduling for DAG tasks. Further, Jiang et al. [26], Yang et al. [27] and Shariati et al. [28] extend the federated approach to the semi-federated scheme, which allow both heavy and light tasks to share certain cores to execute.

The *partitioned* scheme decomposes DAGs to a set of sequential sporadic tasks, and assign each task with a fixed core to execute. Fonseca et al. [29] decomposes DAGs to a set of self-suspending tasks and presents the first response time analysis for DAGs under a fully-partitioned scheme. Casini et al. [14] provides a partitioning algorithm and a fine-grained analysis for non-preemptive DAG tasks. Further, Bado et al. [30], Maia et al. [31] and Hatami et al. [32] propose semi-partitioned schemes for parallel tasks, which allow certain nodes to execute on more than one cores for better schedulability.

However, most of the above works focus on offline methods which do not consider the workload distribution of the system and the scheduled DAGs. Intuitively, with such knowledge the inter-task interference of DAG tasks can be significantly reduced, and hence, leads to better schedulability. In this paper, we propose an novel multi-DAG scheduling method (see Section VII) which specifies the number of cores dedicated to each DAG task in a hyper-period based on the workload distribution of the current system and the DAG task.

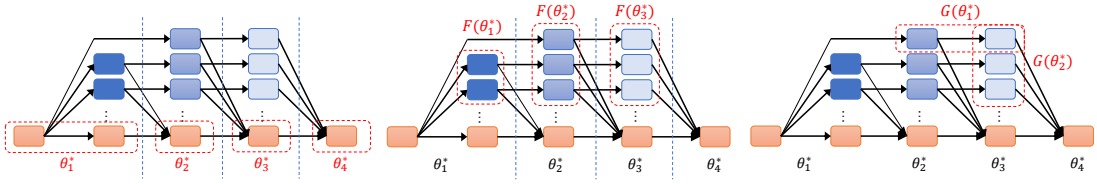
IV. The Concurrent Provider and Consumer Model

Equation 1 indicates that minimising the delay caused by non-critical nodes to the critical path (i.e., $\frac{1}{m}(W-L)$) effectively reduces the makespan of the DAG. Achieving this requires the complete knowledge of the inter-node dependency and node-level parallelism of a DAG so that the potential delay imposed from the non-critical nodes to the critical path can be identified. To support this, the CPC model is proposed to obtain an in-depth understanding of DAG structure and enable full exploitation of node dependency as well as parallelism.

The intuition of the CPC model is: when the critical path is executing, it utilises just one core so that the non-critical ones can execute in parallel on the remaining $(m-1)$ cores. The time allowed for executing non-critical nodes in parallel is termed as the *capacity*. We note that the non-critical nodes that utilise this capacity to execute cannot cause any delay to the critical path. Therefore, by obtaining the knowledge of which non-critical nodes can execute in parallel with a critical node, the CPC model provides valuable information to (i) the analysis so that the parallel workload can be explicitly computed to reduce pessimism and (ii) the scheduling method so that a node execution order can be produced to maximise such parallel workload.

Algorithm 1 presents the process for constructing the CPC model of an input DAG $\mathcal{G}$ with its critical path λ^*. The construction of the CPC model contains two key steps. The first step (lines 2-8) identifies the capacity providers Θ^*. The second step (lines 10-14) identifies the capacity consumers Θ. Function $\lambda^*(n)$ returns the nth node in critical path λ^* based on the topological order, where n starts from one. Notations i and k indicate the index of providers and nodes in λ^*, respectively. Table I summarises notations introduced by the proposed CPC model.

Step One. For a given DAG, the capacity providers are constructed from the nodes in its critical path λ^*. Starting from the head node in λ^*, the first provider θ_1^* is formed by taking nodes in λ^* (following topological order) until a node can be delayed by non-critical ones based on its precedence constraints (line 4-7). Figure 3(a) provides an example for identifying the providers, where θ_1^* only takes the first two critical nodes (i.e., $\theta_1^* = \{\lambda^*(1), \lambda^*(2)\}$) because the third one can be delayed by non-critical ones. The algorithm then continues to



(a) identifying providers $\boldsymbol{\theta_i^*}$ (b) identifying consumer groups $\boldsymbol{F(\theta_i^*)}$ (c) identifying consumer groups $\boldsymbol{G(\theta_i^*)}$

Figure 3: The construction of the CPC model. The critical path is highlighted in orange and the non-critical nodes are in blue. The darker the colour of a non-critical node is, the earlier it can delay the critical path.

Algorithm 1: $CPC(\mathcal{G}, \lambda^*)$: CPC model construction

Inputs : $\{\mathcal{G} = (V, E)\}$; λ^*
Outputs : Θ^*; $F(\theta_i^*), G(\theta_i^*), \forall \theta_j^* \in \Theta^*$
Initialise : $V^\neg = V \backslash \lambda^*$; $\Theta^* = \varnothing$; $i = 1$; $k = 1$

1 /* Step 1: identifying capacity providers */
2 **while** $k \leq |\lambda^*|$ **do**
3 $\quad \theta_i^* = \{\lambda^*(k)\}$; k++;
4 $\quad$ **while** $pre(\lambda^*(k)) = \{\lambda^*(k-1)\}$ **do**
5 $\quad\quad \theta_i^* = \theta_i^* \cup \{\lambda^*(k)\}$; k++;
6 $\quad$ **end**
7 $\quad \Theta^* = \Theta^* \cup \theta_i^*$; i++;
8 **end**
9 /* Step 2: identifying capacity consumers */
10 **for** $i = 1$; $i < |\Theta^*|$; i++ **do**
11 $\quad F(\theta_i^*) = anc(\theta_{i+1}^*) \cap V^\neg$;
12 $\quad G(\theta_i^*) = \bigcup_{v_j \in F(\theta_i^*)} \{\mathcal{C}(v_j) \cap V^\neg\}$;
13 $\quad V^\neg = V^\neg \setminus F(\theta_i^*)$;
14 **end**
15 **return** $\{\Theta^*; F(\theta_i^*), G(\theta_i^*), \forall \theta_i^* \in \Theta^*\}$

construct the second provider θ_2^* by taking the third node in λ^*. This process finishes until all nodes in λ^* are taken as providers.

The principle for constructing the providers is: $\forall \theta_i^* \in \Theta^*$, *once* θ_i^* *starts its execution it should not be delayed by any non-critical node*. By doing so, we guarantee that each provider can (conceptually) provide consecutive capacity without being blocked in the middle of execution.

Step Two. With the providers, the CPC identifies the set of consumers for each provider θ_i^*. Here the CPC classifies the consumers as two types.

The first type of consumers $F(\theta_i^*)$ of provider θ_i^* are identified by the non-critical nodes which (i) can execute concurrently with θ_i^*, and 2) can delay the start of θ_{i+1}^* (i.e., $anc(\theta_{i+1}^*) \cap V^\neg$ in Line 12). That is, nodes in $F(\theta_i^*)$ that finish later than θ_i^* will impose a delay to the start of θ_{i+1}^* (if it exists). Figure 3(b) shows the identification of the consumer group $F(\theta_i)$ of each capacity provider θ_i. For example, nodes in $F(\theta_1^*)$ can delay θ_2^* if they are finished later than θ_1^*. *With* $F(\theta_i^*)$ *identified, the CPC model enables the knowledge of the potential delay caused by non-critical nodes on the critical path.*

The second type of the consumers $G(\theta_i^*)$ (line 13) denotes the nodes that belong to the consumer groups of later providers, but can execute in parallel (in terms of topology) with θ_i^*. For instance, in Figure 3(c) nodes in $G(\theta_1^*)$ can execute in parallel with θ_1^* but cannot directly delay the start of θ_2^* if they finish late. The key difference between $G(\theta_i^*)$ and $F(\theta_1^*)$ is that consumers in $G(\theta_i^*)$ cannot impose a direct delay to θ_{i+1}^*. *However, nodes in* $G(\theta_i^*)$ *can delay the execution of* $F(\theta_i^*)$*, and subsequently, impose an indirect delay on* θ_{i+1}^*.

Table I: Notations introduced in the proposed CPC model.

Notation	Description
Θ^*	The set of capacity providers.
Θ	The set of capacity consumers.
θ_i^*	A capacity provider with an index i.
$F(\theta_i^*)$	The consumer of θ_i^* that can delay θ_{i+1}^*.
$G(\theta_i^*)$	The consumer of θ_i^* that cannot delay θ_{i+1}^*.
$\lambda^*(n)$	The nth node in critical path λ^* by topological order, where n starts from one.

Note, the last provider has an empty set of both $F(\theta_i^*)$ and $G(\theta_i^*)$. This is enforce by the assumption that each DAG has only one sink node (see Section II-A).

Summary. With the CPC model constructed, a DAG is transformed into a set of capacity providers and consumers, with a time complexity of $\mathcal{O}(|V|+|E|)$, i.e., $\mathcal{O}(n)$. For each provider $\theta_i^* \in \Theta^*$, the CPC identifies (i) a set of consumers $F(\theta_i^*)$ that can execute using θ_i^*'s capacity as well as delay the next provider θ_{i+1}^*, and (ii) a set of consumers $G(\theta_i^*)$ that can use θ_i^*'s capacity but will not delay the start of θ_{i+1}^*. For a given provider θ_i^*, nodes in $F(\theta_i^*)$ can utilise a capacity of $len(\theta_i^*)$ on each of $m-1$ cores to execute in parallel while incurring potential delay from $G(\theta_i^*)$. The CPC model enables complete knowledge of both the direct and indirect delay from non-critical nodes to the critical path, and provides the basis for the development of analysis (Section V) and schedule (Section VI) for a single DAG.

V. Parallelism-aware Response Time Analysis

Based on the CPC model, this section presents a new response time analysis, which explicitly accounts for the workload that can execute in parallel with the critical path. The analysis then applies this workload as a safe reduction on the delay of the critical path. Two versions of the analysis are developed.

First, for generality, the proposed analysis assumes the critical path first execution (i.e., CPFE) and allows any scheduling scheme for non-critical nodes. This principle is highlighted by the CPC model and is adopted in many existing methods [18], [1], [13]. Compared to the traditional analysis [16], [17], this analysis provides an improved bound for all schedules based on CPFE. The analysis does not assume the explicit execution order for the non-critical nodes (i.e., consumers) is known in advance.

Then, in Section V-D, we extend the proposed analysis for scheduling methods with an explicit order known a priori (e.g., methods in He[1] and Chen[20]) with minor modifications. Table II summarises the notations introduced in the constructed analysis.

A. The parallel and interfering workload of a provider

From the CPC model we can understand: a provider $\theta_i^* \in \Theta^*$ can start if and only if the previous provider θ_{i-1}^* and its consumers

Table II: Notations introduced in the proposed response time analysis

Notation	Description
p_j	The priority of a node v_j.
$f(\cdot)$	The finish time of a given provider or a consumer node.
L_i	The length of provider θ_i^*.
W_i	The total workload of all nodes in θ_i^*, $F(\theta_i^*)$ and $G(\theta_i^*)$.
α_i	The workload in $F(\theta_i^*)$ and $G(\theta_i^*)$ that can execute in parallel with θ_i^*.
β_i	The *length* of the longest path in $F(\theta_i^*)$ that executes later than $f(\theta_i^*)$.
λ_{v_e}	The *set of nodes* that form the longest path in $F(\theta_i^*)$ that executes later than $f(\theta_i^*)$, with the end node v_e.
Λ_V	Returns all paths of the given input node set V.
$\lvert\cdot\rvert$	returns the size of a given input set.
$\mathcal{I}(v_j)$	The non-critical nodes that can interfere v_j.
$\mathcal{I}^e(\cdot)$	The non-critical nodes that can interfere the input node or path with an explicit execution order.
$I_{\lambda_{v_e},j}$	The actual delay on λ_{v_e} from a node v_j that executed in the interfering workload.

$F(\theta_{i-1}^*)$ have finished executions (see Figure 3(b)). In addition, $F(\theta_{i-1}^*)$ can incur a delay from $G(\theta_{i-1}^*)$ (i.e., early-released consumers that can execute concurrently with $F(\theta_{i-1}^*)$), which in turn, delays the start of θ_i^* (see Figure 3(c)).

To facilitate the construction of the analysis, we formally define the *parallel workload* and *interfering workload* of a capacity provider θ_i^*. The parallel workload can execute along with θ_i^* and the interfering workload may impose a delay to the following providers of θ_i^*.

Let $f(\cdot)$ denote the finish time of a provider θ_i^* or a consumer node v_j, $L_i = len(\theta_i^*)$ gives the length of θ_i^* and $W_i = L_i + \sum_{v_k \in F(\theta_i^*)}\{C_k\} + \sum_{v_k \in G(\theta_i^*)}\{C_k\}$ gives the sum of the workload in θ_i^*, $F(\theta_i^*)$ and $G(\theta_i^*)$. Definitions 1 and 2 define the terms *parallel* and *interfering* workload of θ_i^*, respectively.

Definition 1 (Parallel Workload of θ_i^*)**.** *The parallel workload α_i of θ_i^* is the workload in $W_i - L_i$ that can execute before the time instant $f(\theta_i^*)$.*

For a node v_j in $F(\theta_i^*) \cup G(\theta_i^*)$, it contributes to α_i if either $f(v_j) \leq f(\theta_i^*)$ or $f(v_j) - C_j < f(\theta_i^*)$. The former case (i.e., $f(v_j) \leq f(\theta_i^*)$) indicates v_j is finished before the finish of θ_i^* and cannot cause any delay, whereas $f(v_j) - C_j < f(\theta_i^*)$ means v_j can partially execute in parallel with θ_i^* so that its delay on θ_{i+1}^* is less than C_j. In Section V-C, function $f(\cdot)$ is formulated for both providers and consumers, along with the response time analysis.

Definition 2 (Interfering Workload of θ_i^*)**.** *The interfering workload of θ_i^* is the workload in $W_i - L_i$ that executes after the time instant $f(\theta_i^*)$. For a provider θ_i^*, its interfering workload is $W_i - L_i - \alpha_i$.*

With Definitions 1 and 2, Lemma 1 follows.

Lemma 1. *For providers θ_i^* and θ_{i+1}^*, the workload in W_i that can delay the start of θ_{i+1}^* is at most $W_i - L_i - \alpha_i$.*

Proof. Based on the CPC, the start of θ_{i+1}^* depends on the finish of both θ_i^* and $F(\theta_i^*)$, which is $\max\{f(\theta_i^*), \max_{v_j \in F(\theta_i^*)} f(v_j)\}$. By Definition 1, α_i will not cause any delay as it always finishes before $f(\theta_i^*)$, and hence, the lemma follows. Note that although $G(\theta_i^*)$ cannot delay θ_{i+1}^* directly, it can delay on nodes in $F(\theta_i^*)$, and in turn, causes an indirect delay to θ_{i+1}^*. □

B. The (α, β)-pair analysis formulation

Based on Definitions 1 and 2, the parallel workload α_i of θ_i^* finishes no later than $f(\theta_i^*)$ on $m-1$ cores. After θ_i^* completes, the interfering workload (if any) then executes on all m cores, in which the latest-finished node in $F(\theta_i^*)$ gives the earliest starting time to the next provider (if it exists). Therefore, bounding this delay requires:

1) a bound on the parallel workload (i.e., α_i);
2) a bound on the longest execution sequence in $F(\theta_i^*)$ that executes later than $f(\theta_i^*)$ (i.e., in the interfering workload), denoted as β_i.

With a random execution order, the worst-case finish time of β_i effectively upper bounds the worst-case finish of workload in $F(\theta_i^*)$ that executes later than $f(\theta_i^*)$ [17], [16].

With α_i and β_i defined, Lemma 2 gives the bound on the delay θ_i^* that can incur due to the consumer nodes in $F(\theta_{i-1}^*)$.

Lemma 2. *For two consecutive providers θ_{i-1}^* and θ_i^*, the consumers nodes in $F(\theta_{i-1}^*)$ can delay θ_i^* by at most $\left\lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) + \beta_i \right\rceil$.*

Proof. By Definition 2, the interfering workload in $F(\theta_i^*) \cup G(\theta_i^*)$ that can (directly or transitively) delay θ_{i+1}^* is at most $W_i - L_i - \alpha_i$. Given the longest execution sequence in $F(\theta_i^*)$ in the interfering workload (i.e., β_i), the worst-case finish time of $F(\theta_i^*)$ (and also β_i) is bounded as $\left\lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) \right\rceil + \beta_i$, for a system with m cores. This is proved in Graham[17] and Melani[16]. Note, as β_i is accounted for explicitly, it is removed from the interfering workload to avoid repetition. □

Based on Lemma 2, the response time can be formulated for a DAG task, as shown in Equation 2. As $W_i - L_i - \alpha_i$ starts strictly after $f(\theta_i^*)$ (see Definition 1), the finish time of both θ_i^* and $F(\theta_i^*)$ is bounded by the length of θ_i^* (i.e., L_i) and the worst-case finish time of β_i. In addition, θ_{i+1}^* can only start after the finish of θ_i^* and all nodes in $F(\theta_i^*)$. Thus, the final response time of the DAG is bounded by the sum of the finish time of each provider and its consumers.

$$R = \sum_{\theta_i^* \in \Theta^*} \left\{ L_i + \left\lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) \right\rceil + \beta_i \right\} \tag{2}$$

Compared to the traditional analysis [17], [16], this analysis can improve the worst-case response time approximations, by tightening the interference on the critical path (i.e., α_i), without undermining the correctness of the analysis (i.e., with β_i). In the case of $\left\lceil \frac{1}{m}(W - L) \right\rceil > \sum_{\theta_i^* \in \Theta} \left\lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) \right\rceil + \beta_i$, a tighter bound can be obtained. That is, the proposed analysis does not always dominate the traditional bound. Therefore, we take $\min\{R, L + \left\lceil \frac{1}{m}(W - L) \right\rceil\}$ as the final analytical bound.

C. Bounding α_i and β_i

Notations α_i and β_i can be bounded by examining $f(\theta_i^*)$ and $f(v_k), \forall v_k \in F(\theta_i^*) \cup G(\theta_i^*)$ in the scenario that one core is dedicated to θ_i^* and $(m-1)$ cores can be used by $F(\theta_i^*)$.

For a node v_j, it can be subject to interference (say I_j) from the concurrent nodes upon arrival. Before bounding $f(v_j)$, we first distinguish two special situations in which the interference of a node v_j is zero, as given in Lemma 3, with $\mathcal{C}(v_j)$ gives v_j's concurrent nodes, Λ_V denotes paths in a given node set V and $|\cdot|$ returns the size of a given set.

Lemma 3. *Under a schedule with CPFE, node v_j does not incur any interference from its concurrent nodes $\mathcal{C}(v_j)$, if $v_j \in \lambda^* \vee |\Lambda_{\mathcal{C}(v_j) \setminus \lambda^*}| < m-1$.*

Proof. First, the interference of v_j is zero if $v_j \in \lambda^*$. This is enforced by CPFE (i.e., Rule 1), where a critical node always starts immediately after all nodes in $pre(v_j)$ have finished their executions.

Second, a node $v_j \in V^{\neg}$ does not incur any interference if $|\Lambda_{\mathcal{C}(v_j) \setminus \lambda^*}| < m-1$. The concurrent nodes that can interfere v_j

on $(m-1)$ cores is $\mathcal{C}(v_j)\backslash\lambda^*$. Given that the number of paths in $\mathcal{C}(v_j)\backslash\lambda^*$ is less than $m-1$, at least one core is idle when v_j is ready so that it can start directly with no interference. □

Followed by Lemma 3, Equation 3 provides the bound on $f(v_j), v_j \in V$. For a node v_j, it cannot release until all $v_k \in pre(v_j)$ have completed. This is enforced by the precedence constraints from the DAG structure, and hence $\max_{v_k \in pre(v_j)}\{f(v_k)\}$. In addition, if v_j does not satisfy either case in Lemma 3, v_j can incur an worst-case interference of $\frac{1}{m-1}\sum_{v_k \in \mathcal{I}(v_j)} C_k$, in which $\mathcal{I}(v_j)$ denotes the non-critical nodes that can interfere v_j (see Equation 4) [1]. The condition $|\Lambda_{\mathcal{C}(v_j)\backslash\lambda^*}| < m-1$ is checked by Line 8-9 in Algorithm 2 with $m-1$ searches, which identifies a path in the given node set during each search.

$$f(v_j) = C_j + \max_{v_k \in pre(v_j)}\Big\{f(v_k)\Big\} + \begin{cases} 0, & \text{if } v_j \in \lambda^* \vee |\Lambda_{\mathcal{C}(v_j)\backslash\lambda^*}| < m-1 \\ \Big\lceil \frac{1}{m-1} \times (\sum_{v_k \in \mathcal{I}(v_j)} C_k)\Big\rceil, & \text{otherwise} \end{cases} \tag{3}$$

Equation 3 bounds $f(v_j)$ by recursively computing the finish time of all nodes in $anc(v_j)$. To guarantee each node is taken into account only once when bounding the finish time of v_j, $\mathcal{I}(v_j)$ only takes the concurrent non-critical nodes that cannot delay $anc(v_j)$ [1], as given in Equation 4. Note that this equation only applies to non-critical nodes v_j with $|\Lambda_{\mathcal{C}(v_j)\backslash\lambda^*}| \geq m-1$.

$$\mathcal{I}(v_j) = \Big\{v_k | v_k \notin \lambda^* \wedge v_k \notin \bigcup_{v_l \in anc(v_j)} \mathcal{I}(v_l), \forall v_k \in \mathcal{C}(v_j)\Big\} \tag{4}$$

With $f(v_j), \forall v_j \in V$ computed, the worst-case finish time of a provider θ_i^* and its $F(\theta_i^*)$ can be obtained, as given in Equations 5 and 6 respectively.

$$f(\theta_i^*) = \max_{\forall_{v_j \in \theta_i^*}}\Big\{f(v_j)\Big\} \tag{5}$$

$$f(F(\theta_i^*)) = \max_{\forall_{v_j \in F(\theta_i^*)}}\Big\{f(v_j)\Big\} \tag{6}$$

To this end, α_i and β_i can be effectively upper bounded by examining the $f(\theta_i^*)$ and $f(v_j),\ \forall v_j \in F(\theta_i^*) \cup G(\theta_i^*)$. Equation 7 gives the bound on α_i.

$$\begin{aligned} \alpha_i = &\sum_{v_j \in V_a}\Big\{C_j\Big\} + \sum_{v_j \in V_b}\Big\{f(\theta_i^*) - \Big(f(v_j) - C_j\Big)\Big\}, \\ &\forall v_j \in F(\theta_i^*) \cup G(\theta_i^*), \\ \text{where} \\ V_a = &\{v_j | f(v_j) \leq f(\theta_i^*)\} \\ V_b = &\{v_j | f(v_j) > f(\theta_i^*) \wedge f(v_j) - C_j < f(\theta_i^*)\} \end{aligned} \tag{7}$$

This equation is derived from Definition 1. For $v_j \in F(\theta_i^*) \cup G(\theta_i^*)$, it can contribute to α_i if (i) it finishes before θ_i^*, i.e., $f(v_j) \leq f(\theta_i^*)$, or (ii) it finishes after $f(\theta_i^*)$ but with a start time earlier than $f(\theta_i^*)$, i.e., $f(v_j) > f(\theta_i^*) \wedge f(v_j) - C_j < f(\theta_i^*)$. The former case gives V_a in the equation, with nodes in V_a fully contributing to α_i by C_a. The later case gives the set V_b, in which nodes in V_b are partially contributing to α_i by $f(\theta_i^*) - (f(v_j) - C_j)$.

Then, β_i can be decided by the longest path of $F(\theta_i^*)$ that executed later than $f(\theta_i^*)$, i.e., in the interfering workload. Let λ_{v_e} denote this path ending with node v_e, Lemmas 4 and 5 identifies v_e and its predecessor node in λ_{v_e}, among all nodes in $F(\theta_i^*)$.

Lemma 4. *For the end node v_e in the longest path of $F(\theta_i^*)$, $f(v_e) = f(F(\theta_i^*))$.*

Proof. Given two paths λ_a and λ_b with length $L_a > L_b$ and a total workload of W, it follows $f(\lambda_a) = L_a + \frac{1}{m}(W - L_a) \geq f(\lambda_b) = L_b + \frac{1}{m}(W - L_b)$, as $f(\lambda_a) - f(\lambda_b) = L_a - L_b + \frac{1}{m}(L_b - L_a) \geq 0$. Therefore, node v_e with $f(v_e) = f(F(\theta_i^*))$ gives the end node of the longest path in the interfering workload. □

Lemma 5. *The predecessor node of the end node v_e in the longest path of $F(\theta_i^*)$ is given by $\underset{v_j}{argmax}\{f(v_j) \mid \forall v_j \in pre(v_e) \cap F(\theta_i^*)\}$.*

Proof. Given $v_a, v_b \in pre(v_c)$ with $f(v_a) \geq f(v_b)$, we have $len(\lambda_{v_a} \cup v_c) \geq len(\lambda_{v_b} \cup v_c)$ [1]. Therefore, the predecessor node of v_e with the latest finish is in the longest path ending with v_e in $F(\theta_i^*)$. □

Based on Lemmas 4 and 5, λ_{v_e} is computed recursively by Equation 8. Starting from v_e, λ_{v_e} searches through the predecessor nodes recursively and includes the one with the longest finish time in each recursion, until a complete path is obtained or all predecessors are finished before $f(\theta_i^*)$.

$$\begin{aligned} \lambda_{v_e} = \lambda_{v_j} \cup v_e : \\ \underset{v_j}{argmax}\Big\{f(v_j)\Big|\forall v_j \in pre(v_e) \wedge f(v_j) > f(\theta_i^*)\Big\} \\ \underset{v_e}{arg}\Big\{f(v_e) = f(F(\theta_i^*))\Big\}, \quad v_e, v_j \in F(\theta_i^*) \end{aligned} \tag{8}$$

With λ_{v_e} obtained, β_i is computed by Equation 9, which bounds the workload in λ_{v_e} that is executed later than $f(\theta_i^*)$.

$$\beta_i = \sum_{v_j \in \lambda_{v_e}} \begin{cases} C_j, & \text{if } f(v_j) - C_j \geq f(\theta_i^*) \\ f(v_j) - f(\theta_i^*), & \text{otherwise} \end{cases} \tag{9}$$

For the first node in λ_{v_e} (say v_s), two cases can occur based on its worst-case start time $f(v_s) - C_s$. First, with $f(v_s) - C_s \geq f(\theta_i^*)$, v_s starts after the finish of θ_i^* and fully contributes to the interfering workload. Otherwise (i.e., $f(v_s) - C_s < f(\theta_i^*)$), v_s partially contributes to α_i, i.e., $v_s \in V_b$ in Equation 7. Thus, by Definitions 1 and 2, it can contribute at most $(f(v_s) - f(\theta_i^*))$ to the interfering workload. Note that v_s is the only node in λ_{v_e} that can have $f(v_s) - C_s < f(\theta_i)$.

With α_i and β_i computed for each provider $\theta_i^* \in \Theta^*$, the response time analysis for scheduling methods that feature CPFE is complete.

Sustainability: It is worth noting that this analysis is sustainable, i.e., provides a safe bound if any node executes less than its WCET. We demonstrate this by reducing the WCET of a randomly node in $V^{\neg}$ and λ_i^* respectively [33].

First, suppose $v_j \in F(\theta_i^*) \cup G(\theta_i^*)$ executes less than its WCET, denoted as $C_j' < C_j$. Based on Equation 3, it leads to $f'(\theta_i^*) = f(\theta_i^*)$ as $v_j \notin pre(v_k), \forall v_k \in \theta_i^*$, and $f'(v_k) \leq f(v_k), \forall v_k \in F(\theta_i^*) \cup G(\theta_i^*)$. Based on Definitions 1 and 2, we have $\alpha_i' \geq \alpha_i$ and $\beta_i' \leq \beta_i$. This guarantees there is a non-increasing delay on the start of θ_{i+1}^* based on Equation 2.

Second, if a provider node $v_j \in \theta_i^*$ executes $C_j' < C_j$, we have $f'(\theta_i^*) < f(\theta_i^*)$ and $f'(v_k) = f(v_k), \forall v_k \in F(\theta_i^*) \cup G(\theta_i^*)$ based on Equation 3. That is, the parallel workload obtained by Equation 7 (with full WCET C_j) can still finish before $f(\theta_i^*)$ in the case of C_j', and subsequently, the interfering workload can start no later than $f(\theta_i^*)$ on all m cores. Thus, the finish time of θ_i^* and $F(\theta_i^*)$ cannot exceed the bound obtained by Equation 2 with full WCET.

Combining both, a decrease in WCET of arbitrary nodes in a DAG leads to a non-increasing bound on its completion. Therefore, the proposed analysis provides a safe worst-case bound as long as each node in the DAG does not exceed its WCET, i.e., it is sustainable.

D. Supporting explicit execution order

With an explicit scheduling order for non-critical nodes, a tighter bound can be obtained as each node can only incur interference from concurrent nodes with a higher priority [1]. Using the proposed schedule as an example, this section illustrates a novel analysis that can support CPFE and explicit execution order for non-critical nodes.

With an explicit node ordering, the interfering nodes of v_j on $m-1$ cores can be effectively reduced to i) nodes in $\mathcal{I}(v_j)$ that have a higher priority than p_j [1], and ii) $m-1$ nodes in $\mathcal{I}(v_j)$ that have a lower priority and the highest WCET due to the non-preemptive schedule [34]. Let $\mathcal{I}^e(v_j)$ denote the nodes that can interfere a non-critical node v_j with an explicit order, it is given as Equation 10, in which $\text{argmax}_{v_k}^{m-1}$ returns the first $m-1$ nodes with the highest value of the given metric (C_k in this equation). The correctness of the equation is proven in He[1] and Serrano[34]. For simplicity, we take the $(m-1)$ low priority nodes as a safe upper bound. A more accurate ILP-based approach is available in Serrano[34] to precisely compute this blocking. In addition, if node-level preemption is allowed, $\mathcal{I}^e(v_j)$ is further reduced to $\{v_k | p_k > p_j, v_k \in \mathcal{I}(v_j)\}$.

$$\mathcal{I}^e(v_j) = \Big\{ v_k | p_k > p_j, v_k \in \mathcal{I}(v_j)\} \cup \underset{v_k}{\overset{m-1}{argmax}} \{C_k | p_k < p_j, v_k \in \mathcal{I}(v_j)\Big\} \tag{10}$$

With this schedule, $f(v_j), \forall v_j \in V$ can be computed by Equation 3, with $\mathcal{I}^e(v_j)$ applied to non-critical nodes executing on $m-1$ cores. Hence, α_i and β_i can be bounded with the updated $f(\theta_i^*)$ and $f(v_j),\ \forall v_j \in F(\theta_i^*) \cup G(\theta_i^*)$, by Equation 7 and 9 respectively. Note that with an explicit schedule, λ_{v_e} computed in Equation 8, it is not necessarily the longest path in $F(\theta_i^*)$ that executes in the interfering workload [1]. Instead, λ_{v_e} in this case gives the path that will always finish last due to the pre-planned node execution order.

The final bound on the response time of the DAG task is, however, different from the generic case, i.e., Equation 2. With node priority, it is not necessary that all workload in $(W_i - L_i - \alpha_i - \beta_i)$ can interfere with the execution of λ_{v_e}. Let R^e denote the response time of a DAG task with an explicit scheduling order. It is bound in Equation 11, in which $\mathcal{I}^e(\lambda_{v_e})$ determines the nodes that can delay λ_{v_e} and $I_{\lambda_{v_e},j}$ gives the actual delay on λ_{v_e} from node v_j in the interfering workload.

$$R^e = \sum_{\theta_i^* \in \Theta^*} L_i + \beta_i + \begin{cases} 0, & \text{if } |\Lambda_{\mathcal{I}^e(\lambda_{v_e})}| < m \\ \left\lceil \frac{1}{m} \times \sum_{v_j \in \mathcal{I}^e(\lambda_{v_e})} I_{\lambda_{v_e},j} \right\rceil, & \text{otherwise} \end{cases} \tag{11}$$

Given the length of θ_i^* (L_i) and the worst-case delay on λ_{v_e} ($\mathrm{I}_{\lambda_{v_e}}$) in the interfering workload, the worst-case finish time of θ_i^* and $F(\theta_i^*)$ is upper bounded by $L_i + \beta_i + \left\lceil \frac{1}{m} \times \sum_{v_j \in \mathcal{I}^e(\lambda_{v_e})} I_{\lambda_{v_e},j} \right\rceil$. This is proved in Lemma 2. In addition, if the number of paths in the nodes that can cause $\mathrm{I}_{\lambda_{v_e}}$ is less than m (i.e., $|\Lambda_{\mathcal{I}^e(\lambda_{v_e})}| < m$), λ_{v_e} executes directly after θ_i^* and finishes by $L_i + \beta_i$. This is proved in Lemma 3. Note that $\mathrm{I}_{\lambda_{v_e}} = 0$ if $\beta_i = 0$, as all workload in $F(\theta_i^*)$ contributes to α_i so that θ_{i+1}^* (if it exists) can start immediately after θ_i^*.

The nodes that can interfere with λ_{v_e} (i.e., $\mathcal{I}^e(\lambda_{v_e})$) are given by Equation 12, in which $I_{\lambda_{v_e},j}$ gives the actual delay from node v_j on λ_{v_e}.

$$\mathcal{I}^e(\lambda_{v_e}) = \bigcup_{v_k \in \lambda_{v_e}} \Big\{ v_j | f(v_j) > f(\theta_i^*) \wedge p_j > p_k, \forall v_j \in \mathcal{I}(v_k) \Big\} \cup \bigcup_{v_k \in \lambda_{v_e}} \underset{v_k}{\overset{1..m}{argmax}} \Big\{ I_{\lambda_{v_e},j} | f(v_j) > f(\theta_i^*) \wedge p_j < p_k, v_j \in \mathcal{I}(v_k) \Big\} \tag{12}$$

Finally, $I_{\lambda_{v_e},j}$ is bound by Equation 13, which takes the workload of v_j executed after $f(\theta_i^*)$ (i.e., in the interfering workload) as the worst-case delay on λ_{v_e}.

$$I_{\lambda_{v_e},j} = \begin{cases} C_j, & \text{if } f(v_j) - C_j \geq f(\theta_i^*) \\ f(v_j) - f(\theta_i^*), & \text{otherwise} \end{cases} \tag{13}$$

This concludes the response time analysis constructed in this paper. The analysis can be applied to scheduling methods with node execution order known a priori. As with the generic bound, this analysis is sustainable, as a reduction in WCET of any arbitrary node cannot lead to completion later than the worst-case bound (see Section IV). Compared to the generic bound for non-critical nodes with random order, this analysis provides tighter results by removing the nodes that cannot cause a delay due to their priorities, in which $\mathcal{I}^e(v_j) \subseteq \mathcal{I}(v_j)$ and $\mathrm{I}_{v_e} \leq W_i - L_i - \alpha_i - \beta_i$.

VI. DAG Scheduling: A Parallelism and Node Dependency Exploited method

In this section, a scheduling method is developed to reduce DAG makespan during execution, by maximising node parallelism based on the CPC model. This is achieved by a rule-based priority assignment, in which three rules are developed to statically assign a priority to each node in the DAG. The Rule 1 (in Section VI-A) always executes the critical path first (i.e., the longest path in a DAG), then Rules 2 and 3 (Section VI-B) maximise parallelism and minimise the delay to the critical path. The entire proposed approach has general applicability to DAGs with any topology (unlike, e.g., the schedule Fonseca[13], which assumes nested fork-join DAGs only). It assumes a homogeneous architecture, however, it is not restricted by the number of processors.

The example DAG in Figure 2 is used throughout the presentation of the constructed priority assignment to illustrate its key rationale. With CPC applied, its critical path forms three providers $\theta_1^* = \{v_1, v_5\}$, $\theta_2^* = \{v_7\}$ and $\theta_3^* = \{v_8\}$, where the delay from non-critical nodes only occurs on the head node of the providers. For each provider, we have $F(\theta_1^*) = \{v_6\}$, $F(\theta_2^*) = \{v_2, v_3, v_4\}$ and $F(\theta_3^*) = \varnothing$. In addition, all nodes in $F(\theta_2^*) = \{v_2, v_3, v_4\}$ can start earlier than θ_2^* delaying the execution of $F(\theta_1^*)$ and subsequently, the start of θ_2^*. Therefore, $G(\theta_1^*) = \{v_2, v_3, v_4\}$ and $G(\theta_2^*) = G(\theta_3^*) = \varnothing$.

A. The "Critical Path First" execution

In the CPC model, the critical path is conceptually modelled as a set of capacity providers. Arguably, each complete path can be seen as the providers, which offers the time interval of its path length for other nodes to execute in parallel. However, the critical path provides the maximum capacity and hence, enables the maximised total parallel workload (denoted as $\alpha = \sum_{\theta_i^* \in \Theta^*} \alpha_i$). This provides the foundation to minimise the interfering workload on the complete critical path.

Theorem 1. *For a schedule $\mathcal{S}$ with CPFE and a schedule $\mathcal{S}'$ that prioritises a random complete path over the critical path, the total parallel workload of providers in S is always equal to or higher than that of S', i.e., $\alpha \geq \alpha'$.*

Proof. The change from $\mathcal{S}$ to $\mathcal{S}'$ leads to two effects: (i) a reduction on the length of the provider path, and (ii) an increase on length of one consumer path. Below we prove both effects cannot increase the parallel workload after the change.

First, suppose the length of provider θ_i^* is shortened by Δ after the change from $\mathcal{S}$ to $\mathcal{S}'$, the same reduction applies on its finish time, i.e., $f'(\theta_i^*) = f(\theta_i^*) - \Delta$. Because nodes in θ_i^* are shortened, the

finish time $f(v_j)$ of a consumer node $v_j \in F(\theta_i^*) \cup G(\theta_i^*)$ can also be reduced by a value from Δ/m (i.e., a reduction on v_j's interference, if all the shortened nodes in θ_i^* belong to $\mathcal{C}(v_j)$) to Δ (if all such nodes belong to $pre(v_j)$) [17], [16]. By definition 1, a consumer $v_j \in F(\theta_i^*) \cup G(\theta_i^*)$ can contribute to the α_i if $f(v_j) \leq f(\theta_i^*)$ or $f(v_j) - C_j \leq f(\theta_i^*)$. Therefore, α_i cannot increase in $\mathcal{S}'$, as the reduction on $f(\theta_i^*)$ (i.e., Δ) is always equal or higher than that of $f(v_j)$ (i.e., Δ/m or Δ).

Second, let L and L' denote the length of the provider path under $\mathcal{S}$ and $\mathcal{S}'$ (with $L \geq L'$), respectively. The time for non-critical nodes to execute in parallel with the provider path is L' on each of $m-1$ cores under $\mathcal{S}'$. Thus, a consumer path with its length increased from L' to L directly leads to an increase of $(L - L')$ in the interfering workload, as at most L' in the consumer can execute in parallel with the provider.

Therefore, both effects cannot increase the parallel workload after the change from $\mathcal{S}$ to $\mathcal{S}'$, and hence, $\alpha \geq \alpha'$. □

Theorem 1 leads to the first assignment rule that assigns critical nodes with the highest priority, where p_j is the priority of node v_j.

Rule 1. $\forall v_j \in \Theta^*, \forall v_k \in \Theta \Rightarrow p_j > p_k$.

With Rule 1, the maximum parallel capacity is guaranteed so that an immediate reduction (i.e., α) on the interfering workload of λ^* can be obtained. For the example DAG in Figure 2, Rule 1 leads to the execution scenarios with a makespan of 16 and 13, and avoids the worst case. In Section V, an analytical bound on α_i for each provider θ_i^* is presented, with consumers nodes executed either randomly or under an explicit schedule.

B. Exploiting parallelism and node dependency

With CPFE, the next objective is to maximise the parallelism of non-critical nodes and reduce the delay on the completion of the critical path. Based on the CPC model, each provider θ_i^* is associated with $F(\theta_i^*)$ and $G(\theta_i^*)$. For $v_j \in G(\theta_i^*)$, it can execute before $F(\theta_i^*)$ and use the capacity of θ_i^* to execute, if assigned with a high priority. Under this case, v_j can 1) delay the finish of $F(\theta_i^*)$ and the start of θ_{i+1}^*, and 2) waste the capacity of its own provider. A similar observation is also obtained in He[1], which avoids this delay by the heuristic of early interference node first.

Therefore, the second assignment rule is derived to specify the priority between consumer groups of each provider. For any two adjacent providers θ_i^* and θ_{i+1}^*, the priority of any consumer in $F(\theta_i^*)$ is higher than that of all consumers in $F(\theta_{i+1}^*)$.

Rule 2. $\forall \theta_i^*, \theta_l^* \in \Theta^* : i < l \Rightarrow \min_{v_j \in F(\theta_i^*)} p_j > \max_{v_k \in F(\theta_l^*)} p_k$.

With Rule 2, the delay from $G(\theta_i^*)$ on $F(\theta_i^*)$ (and hence θ_{i+1}^*) can be minimised, because all nodes in $G(\theta_i^*)$ belong to consumers of the following providers and are always assigned with a lower priority than nodes in $F(\theta_i^*)$. With Rules 1 and 2 applied to the DAG in Figure 2, the delay from v_6 on the critical path can be avoided, by assigning v_6 with a higher priority than that of $\{v_2, v_3, v_4\}$.

We now schedule the consumer nodes in each $F(\theta_i^*)$. In He[1], concurrent nodes with the same earliness (in terms of the time they become ready during the execution of the critical path) are ordered by the length of their longest complete path (i.e., from v_{src} to v_{sink}). However, based on the CPC model, a complete path can be divided into several local paths, each of these local paths belongs to the consumer group of different providers. For local paths in $F(\theta_i^*)$, the order of their lengths can be the exact opposite to that of their complete paths. Therefore, this approach can lead to a prolonged finish of $F(\theta_i^*)$.

Algorithm 2: $EA(\Theta^*, \Theta)$: Priority Assignment

Inputs : $\Theta^*; \Theta$
Outputs : $p_j, \forall v_j \in \Theta^* \cup \Theta$
Initialise : $p_j = -1, \forall v_j \in \Theta^* \cup \Theta$; $p = p^{max}$

1 /* Assignment Rule 1. */
2 $\forall v_j \in \Theta^*, p_j = p; \quad p = p - 1;$
3 /* Assignment Rule 2. */
4 **for** each $\theta_i^* \in \Theta^*$, in topological order **do**
5 **while** $F(\theta_i^*) \neq \varnothing$ **do**
6 /* Find the longest local path in $F(\theta_i^*)$. */
7 $v_e, v_j \in F(\theta_i^*)$:
8 $v_e = \underset{v_e}{argmax}\{l_e(F(\theta_i^*)) | suc(v_e) = \varnothing\};$
9 $\lambda_{v_e} = v_e \cup \lambda_{v_j},\ \underset{v_j}{argmax}\{l_j(F(\theta_i^*)) | \forall v_j \in pre(v_e)\};$
10 **if** $|pre(v_j)| > 1, \exists v_j \in \lambda_{v_e}$ **then**
11 $\{\Theta^{*'}, \Theta'\} = CPC(F(\theta_i^*), \lambda_{v_e});$
12 $EA(\Theta^{*'}, \Theta');$
13 **break**;
14 **else**
15 /* Assignment Rule 3. */
16 $\forall v_j \in \lambda_{v_e}, p_j = p; \quad p = p - 1;$
17 $F(\theta_i^*) = F(\theta_i^*) \setminus \lambda_{v_e};$
18 **end**
19 **end**
20 **end**
21 **return** $\{p_j, \forall v_j \in \Theta^* \cup \Theta\}$

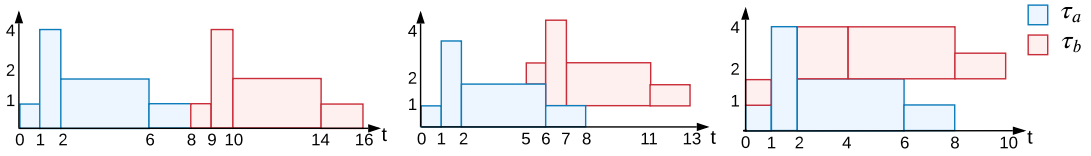
In the constructed schedule, we guarantee a longer local path is always assigned with a higher priority in a dependency-aware manner. This derives the final assignment rule, as given below.

Rule 3*. $v_j, v_k \in F(\theta_i^*) : l_j(F(\theta_i^*)) > l_k(F(\theta_i^*)) \Rightarrow p_j > p_k$

Notation $l_j(F(\theta_i^*))$ denotes the length of the longest local path in $F(\theta_i^*)$ that includes v_j. This length can be computed by traversing $anc(v_j) \cup des(v_j)$ in $F(\theta_i^*)$ [1]. For example, we have $l_2(F(\theta_2^*)) = 7$ and $l_3(F(\theta_2^*)) = l_4(F(\theta_2^*)) = 3$ for the DAG in Figure 2, so v_2 is assigned a higher priority than v_3 and v_4. With Rules 1-3 applied to the example DAG, it finally leads to the best-case schedule with a makespan of 13.

However, simply applying Rule 3 to each $F(\theta_i^*)$ is not sufficient. Given a complex DAG structure, every $F(\theta_i^*)$ can form a smaller DAG $\mathcal{G}'$, and hence, an inner nested CPC model with the longest path in $F(\theta_i^*)$ is the provider. Furthermore, this procedure can be recursively applied to keep constructing inner CPC models for each consumer group in a nested CPC model, until all local paths in a consumer group are fully independent. For each inner nested CPC model, Rules 1 and 2 should be applied for maximised capacity and minimised delay of each consumer group, whereas Rule 3 is only applied to independent paths in a consumer group for maximised parallelism (and hence, the star mark on Rule 3). This enables complete awareness of inter-node dependency and guarantees the longest path first in each nested CPC model.

Algorithm 2 provides the complete approach of the rule-based priority assignment, where p^{max} denotes the highest priority of all nodes in the DAG. The method starts from the outer-most CPC model ($CPC(\mathcal{G}, \lambda^*)$), and assigns all provider nodes with the highest priority based on Rule 1 (Line 2). By Rule 2, the algorithm starts from



(a) non-work-conserving schedule in [10] (b) "fan-in" schedule in [10] (c) a schedule with lower delay

Figure 4: Scheduling examples for two DAG tasks τ_a and τ_b. The y-axis indicates the number of cores used by the tasks.

the earliest $F(\theta_i^*)$ (Line 4) and finds the longest local path λ_{v_e} in $F(\theta_i^*)$ (Line 8-9). If there exists dependency between nodes in λ_{v_e} and $F(\theta_i^*)\backslash\lambda_{v_e}$ (Line 9), $F(\theta_i^*)$ is further constructed as an inner CPC model with the assignment algorithm applied recursively (Line 11-12). This resolves the detected dependency by dividing λ_{v_e} into a set of providers. Otherwise, λ_{v_e} is an independent local path so that priority is assigned to its nodes based on Rule 3. The algorithm then continues with $F(\theta_i^*)\backslash\lambda_{v_e}$. The process continues until all nodes in V are assigned with a priority.

The time complexity of Algorithm 2 is quadratic. At most, $|V|+|E|$ calls to Algorithm 1 are invoked to construct the inner CPC models (Line 11), which examines each node and edge in the DAG. Mutually exclusively, Lines 16-17 assign each node with a priority value. Given that the time complexity of Algorithm 1 is $\mathcal{O}(|V|+|E|)$, we have the time complexity $\mathcal{O}((|V|+|E|)^2)$ for Algorithm 2, i.e., $\mathcal{O}(n^2)$. Although Algorithm 2 is recursive, this result holds as a node assigned with a priority will be removed from further iterations (Line 17), i.e., each node (edge) is processed only once.

With the CPC model and the schedule, the complete process for scheduling a DAG consists of three phases: i) transferring the DAG to CPC; ii) statically assigning a priority to each node by the rule-based priority assignment, and iii) executing the DAG by a fixed-priority scheduler. With the input DAG known a priori, phases i) and ii) can be performed offline so that the scheduling cost at run-time is identical to that of the traditional fixed-priority scheduling, which is in widespread use in real-time systems [35].

VII. Offline Core Assignment for Scheduling Multi-DAG Systems

The above sections target a single recurrent DAG task, in which a node in the DAG can only incur the *intra-task* interference from nodes in the same DAG. In a multi-DAG system, a node can incur both *intra-task* interference and *inter-task* interference, where the later one is caused by other DAGs running in parallel. For multi-DAGs, the existing bound in Graham[17] and Melani[16], and He[1] of the inter-task interference is shown in Equation 14. Notation I_x denotes the inter-task interference of τ_x and $hp(x)$ returns DAGs with a priority higher than that of τ_x. The sum of the DAG makespan (i.e., R_x in Equation 1) and its inter-task interference (i.e., I_x) gives the worst-case response time of τ_x on a multi-DAG system [1].

$$I_x = \frac{1}{m} \sum_{\tau_j \in hp(x)} \left\lceil \frac{R_x}{T_j} \right\rceil \cdot W_j \tag{14}$$

This analysis can be pessimistic as it does not take the parallel degree of the DAGs into account. In an extreme case where there is a sufficient number of cores to accommodate the maximum parallel degree of the system, all DAGs can execute when they are ready without incurring any inter-task interference. However, the above analysis still impose a certain amount of inter-task interference on τ_x, as it accounts for this interference based on the sum of the workload of all high priority DAGs released during R_x. This pessimism can become significant when the high priority DAG τ_j has a very large workload W_j, e.g., with very long nodes.

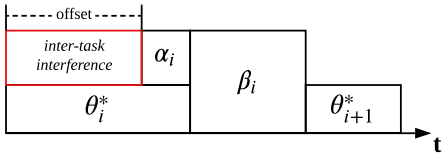


Figure 5: An example illustrating the "offset" effect on the analysis.

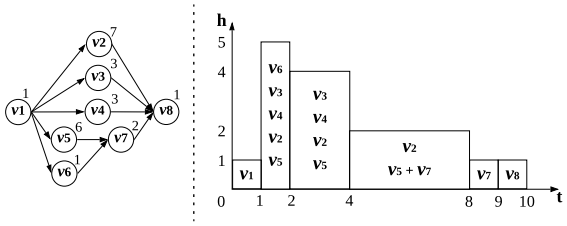
For the proposed analysis in Section V, the inter-task interference brings significant barriers for the analysis to be applied on multi-DAG systems. The key reason is that for a given DAG τ_x, its nodes (especially the non-critical ones) can often incur an additional delay due to other DAGs so that it creates an "offset" effect between the execution of the critical path and the non-critical workload inside τ_x. Figure 5 illustrates the impact of the "offset" on the constructed analysis. With inter-task interference, the execution of the consumer group (i.e., $F(\theta_i^*)$) is delayed so that most of its workload is executed after the provider θ_i^* is finished. For the proposed analysis, this leads to a small α_i yet a large β_i, and subsequently, a large delay on θ_{i+1}^* (see Equation 7). This is because the analysis can only provide fine-grained computations for nodes in the same DAG, and hence, results in a small α_i even though the provider is executed in parallel with other DAGs. Accordingly, the "offset" effect directly breaks the fundamental advantage of the proposed analysis, which then often leads to a high degree of pessimism of the resulting timing bound.

To address the offset effect of the proposed analysis on multi-DAG systems, Zhao[10] presents a non-work-conserving schedule which only allows one DAG to execute at a time and completely avoids inter-task interference. However as shown in Figure 4(a), it can impose a large delay on τ_b, which has to wait for τ_a to finish even if most of the cores are free. Further, based on the non-working-conserving method, Zhao[10] briefly describes a possible "fan-in" schedule which allows τ_b to start (where possible) when τ_a's parallel degree is monotonically non-increasing (i.e., fan-in). As shown in Figure 4(b), τ_b under the "fan-in" schedule is able to start (and finish) 3 units of time earlier than that in Figure 4(a). However, as shown in the figure this method does not address the fundamental challenge and still imposes a large delay to τ_b. The above two scheduling methods indicate that the cost of avoiding the offset effect is the prolonged delay for incoming DAGs. However, as shown in Figure 4(c), if τ_b only utilises two cores to execute, it can (i) avoid the offset effect as the number of cores requested by τ_b is satisfied during its entire execution, and (ii) significantly reduces the delay as the competition on cores is reduced as τ_b now require less cores to execute. With this schedule, τ_b can finish at time 10, with a total delay of only one unit of time.

Based on the motivational example in Figure 4, we propose an

Table III: Notations in scheduling and analysing multi-DAGs.

Notation	Description
τ_x	a DAG task with index x.
$J_{x,j}$	the jth release of τ_x.
P_x	the priority of DAG task τ_x.
I_x	the inter-task interference of τ_x.
$r_{x,j}$	the release time of the jth release of τ_x.
$d_{x,j}$	the deadline of the hth release of τ_x.
$nop_{x,j}$	the number of cores a job $J_{x,j}$ can execute on.
I_x	the inter-task interference of τ_x.
WD_x^m	the P-WDM of τ_x on m cores.
$WD_x^m(t)$	the parallel degree of WD_x^m at time t.
$hp(x)$	the set of tasks with a priority higher than that of τ_x.
$lp(x)$	the set of tasks with a priority lower than that of τ_x.
$A \oplus B$	the accumulation operator of two P-WDMs.



(a) Example DAG (b) Parallel Distribution Model

Figure 6: Workload distribution model on infinite number of cores.

offline core assignment that address the difficulties of applying the analysis on multi-DAG systems and reduces the delay for the incoming DAGs. Essentially, this method allows the parallel execution of multi-DAGs by assigning a specific number of cores to each DAG and (analytically) guarantees that the assigned cores are always available when a DAG starts or resumes execution. By doing so, our analysis can be applied without suffering from the pessimism due to the offset effect. Assigning an appropriate number of cores to each DAG based on its parallelism becomes crucial to the performance of this method.

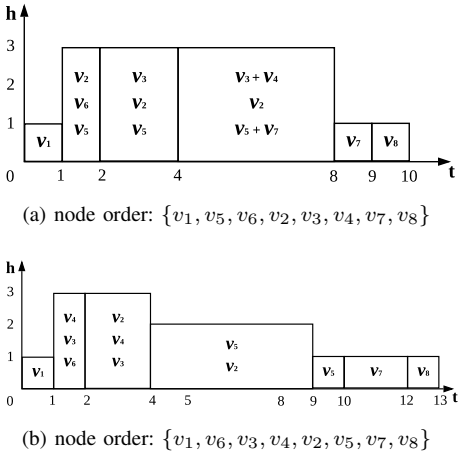
To achieve this, we first propose a Parallelism-aware Workload Distribution (P-WDM) model to understand the internal parallelism of a DAG when running on a any given number of cores (Section VII-B). Then, an P-WDM accumulation method that adds two P-WDMs is developed to understand the workload distribution of the system when multiple DAGs are running in parallel (Section VII-C). Based on this model, the proposed method determines the number of cores that an incoming DAG can run on such that (i) this DAG can execute without suffering from the offset effect due to the currently-executing DAGs and (ii) the deadline of the DAG can be met (Section VII-D). Table III presents notations introduced in this section.

A. Overview of the multi-DAG schedule

As described in Section II, the system contains n sporadic DAG tasks, and each of them can give rise to a set of jobs in a hyper-period. With the multi-DAG schedule constructed in this section, each DAG job $J_{x,j}$ is assigned with a number of cores it can execute on during run-time by the offline core assignment constructed in this section, denoted as $nop_{x,j}$. During run-time, $J_{x,j}$ can be scheduled in a work-conserving manner on $nop_{x,j}$. That is, the scheduler will not assign more cores than $nop_{x,j}$ to $J_{x,j}$, but allows $J_{x,j}$ to execute if there is fewer cores available. In Section VII-D we prove that this does not jeopardise the proposed method and the analysis results.

B. The parallelism-aware workload distribution model

As described above, the key of the proposed method is the effective assignment of the number of cores that each DAG can utilise to execute on. This can impose a direct effect on the parallel degree of the DAG during execution, and subsequently, its finish time. Therefore, it is crucial to understand the execution behaviours (in terms of parallelism) of DAG tasks when executing on different numbers of cores.



(a) node order: $\{v_1, v_5, v_6, v_2, v_3, v_4, v_7, v_8\}$

(b) node order: $\{v_1, v_6, v_3, v_4, v_2, v_5, v_7, v_8\}$

Figure 7: The workload distribution of the example DAG when executing on three cores with two different orders.

The original Workload Distribution Model (WDM) in Fonseca[13] describes the parallel degree of a DAG task on the infinite number of cores. The WDM maps a DAG task to a set of consecutive *execution blocks*, where the height of an execution block denotes the number of nodes that are executing in parallel while its length gives the execution time of the block. Figure 6 presents the WDM of the example DAG task used in Figure 2, where the y-axis shows the number of cores being used by the DAG task with the passage of time (the x-axis).

The WDM can be formally described by Equations 15 and 16. In Equation 15, the start and finish time of each node in a DAG can be computed. For a DAG task that starts at time $t = 0$ with an infinite number of cores, $f^\infty(v_j)$ returns the finish time of a given node v_j in the DAG. Note that on the infinite number of cores, v_j does not incur any interference and can start as long as its predecessors ($pred(v_j)$)) are finished. Then, for a given time instant t, Equation 16 returns the parallel degree of the DAG task, by counting the number of nodes that are actively executing at the time. Notation $\mathcal{PD}^\infty(V_x, t)$ denotes the parallel degree of DAG τ_x at time t, with an infinite number of cores.

$$f^\infty(v_j) = C_j + \max_{v_k \in pred(v_j)} \left(f^\infty(v_k) \right) \tag{15}$$

$$\mathcal{PD}^\infty(V_x, t) = \sum_{v_j \in V_x} \begin{cases} 1, & \text{if } t \in \left[f^\infty(v_j) - C_j, f^\infty(v_j) \right) \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

However, one major limitation of the original WDM is that it cannot describe the parallel distribution of a DAG *when the number of cores is less than the maximum parallelism of the DAG*. To address this limitation, this paper proposes a *Parallelism-aware Workload Distribution Model*, namely P-WDM. The essential difference between the WDM and the P-WDM is that P-WDM can describe the parallel execution of the DAG on any given number of cores. When a DAG is assigned with an insufficient number of cores, the node ordering can impose a direct impact on the shape of workload distribution of the DAG.

Figure 7 presents the workload distribution of the example DAG on three cores with different node orderings. In Figure 7(a), the nodes

Algorithm 3: Construction of the P-WDM

Inputs : V_x; m
Outputs : WD_x^m
Initialise : $t = 0$; $WD_x^m = \varnothing$

1 /* Compute finish time of each node under m cores. */
2 **for** each v_i in V_x **do**
3 | compute $f(v_i)$ by Equation 3;
4 **end**
5 /* Find active nodes at each time instant. */
6 **while** $PD(V_x, t) > 0$ **do**
7 | $WD_x^m = WD_x^m \cup PD(V_x, t)$;
8 | $t = t + 1$;
9 **end**
10 **return** $\{WD_x^m\}$

Algorithm 4: The $\oplus$ operator: accumulation of P-WDMs

Inputs : $WD_a^{m_1}$; $WD_b^{m_2}$; m
Outputs : WD^m; I_b
Initialise : $WD^m = WD_a^{m_1}$; $I_b = 0$

1 **while** $WD_b^{m_2}(t) > 0$ **do**
2 | **if** $WD_a^{m_1}(t) + WD_b^{m_2}(t) \leq m$ **then**
3 | | $WD^m(t) = WD_a^{m_1}(t) + WD_b^{m_2}(t)$;
4 | **else**
5 | | $I_b = I_b + 1$;
6 | **end**
7 | $t = t + 1$;
8 **end**
9 **return** $\{WD^m; I_b\}$

are ordered by the method proposed in Section VI and the workload distribution of the DAG is evenly distributed on three cores. However, a different node ordering would change the workload distribution, e.g., in Figure 7(b) the workload distribution is prolonged by 3 units of time with a lower parallel degree. More importantly, such differences can directly affect the offline core assignment algorithm. For instance, the P-WDM in Figure 7(b) indicates that another incoming DAG with one or two cores assigned can start earlier with less delay.

Therefore, the proposed P-WDM features (i) the ordering at the node level and (ii) the number of cores assigned to the DAG, and describes the workload distribution of the DAG based on these two factors. This can be achieved by Equation 3, which computes the start and finish time of a node v_i, given the node priority and the number of cores. Then, Equation 16 is applied to compute the number of nodes that are executing for a given time instant, but with the notation ∞ removed to feature any given m.

The process of constructing P-WMD is similar to that of the original WMD [13], as given in Algorithm 3. The algorithm takes all nodes in τ_x (i.e., V_x) and a number of cores (i.e., m) as the input, and produces the P-WDM of τ_x on m cores, denoted by WD_x^m. The algorithm first iterates through each node in V_x and computes its finish time $f(v_i)$ (line 1-3). Then, starting from time $t = 0$ (relative to the release of the DAG), the algorithm iterates $PD(V_x, t)$ for each time instance t and adds the current parallel degree of the DAG into WD_x^m (line 4-7). The algorithm terminates until $PD(V_i, t) = 0$, i.e., the finish of the sink node (line 4). During each iteration, it adds the current parallel degree of the DAG into WD_x^m (line 5).

For instance, the P-WDM shown in Figure 7(a) is described as $WD_x^3 = \{1, 3, 3, 3, 3, 3, 3, 3, 1, 1\}$, in which the index of the array gives the relative time t and the value of the array gives the workload distribution of the DAG at t. Please note the time t here denotes the time relative to the release of the DAG, i.e., it always starts from zero even the DAG is not released from the beginning of a hyper-period.

The time complexity of this algorithm is linear, i.e., $\mathcal{O}(n)$, as at most $|V_x|+W_x$ iterations (in the worst case) are required to examine each node in the DAG task. In addition, there exists several implementation approaches to reduce the computation time of this algorithm. For example, the P-WDM can be constructed by iterating the nodes in V_x instead of the relative time t. This will then only require at most $|V_i|\times 2$ iterations which is often much lesser than L_i.

C. The P-WDM of parallel DAGs

When two DAGs are running simultaneously, their P-WDMs can be combined to form a new P-WDM. For the proposed core assignment method, it is important to understand the accumulated workload distribution of the system at a given time so that the method can decide the number of cores that the next DAG can run on without interfering with the existing ones.

To achieve this, we define a new operator $\oplus$ to denote the accumulation of two P-WDMs, where $WD_a^{m_1} \oplus WD_b^{m_2}$ gives a new P-WDM by adding the P-WDM $WD_b^{m_2}$ on top of $WD_a^{m_1}$ under m cores, with $m_1 \leq m$ and $m_2 \leq m$. That is, the operator $\oplus$ features the notion of the non-preemptive schedule, where task τ_a is scheduled before τ_b and cannot be preempted. In addition, the workload accumulation method follows two principles when adding workload:

- The workload distribution of $WD_a^{m_1}$ will not be interfered by the workload in $WD_b^{m_2}$, featuring τ_a has a higher execution eligibility in a non-preemptive scheduling scheme.
- The workload of $WD_b^{m_2}$ will only be added if there exist m_2 free cores, featuring that the τ_b does not incur the offset effect due to core competition.

In this section, the working mechanism of the workload accumulation method is described and the algorithm is presented. Then, we use two illustrative examples to demonstrate the accumulation method $\oplus$ can (i) provide the key to address the issue of applying the proposed analysis on multi-DAG systems and (ii) highlight the intuition of the offline core assignment algorithm.

Algorithm 4 presents the pseudo-code for adding two P-WDMs. The algorithm takes the following inputs: (i) the base P-WDM $WD_a^{m_1}$, (ii) the P-WDM to be added $WD_b^{m_2}$, and (iii) the total number of cores of the system m. It then produces two outputs: (i) the accumulated P-WDM on m cores (WD^m) and (ii) the delay (denoted by I_b) that the task of $WD_b^{m_2}$ incurs when adding its workload on $WD_a^{m_1}$. The algorithm iteratively calculates the workload in $WD_b^{m_2}$ by time t (line 1) and checks whether there exist enough cores to accommodate the workload (line 2). If so, the workload is added on the top of $WD_a^{m_1}$ (line 3). Otherwise, τ_b incurs a delay until the system has enough free cores to allocate the workload (line 5). The algorithm terminates when $WD_b^{m_2}(t) = 0$, which indicates the finish of its sink node. We note that a simple mechanism should be supported to transform the absolute time of the system (which starts from the beginning of the hyper-period) to the relative time of a job (which starts from the execution of the job) when accessing its P-WDM. By performing Algorithm 4 iteratively on the P-WDMs of

all jobs in one hyper-period, one can obtain the complete workload distribution model of the system.

As with Algorithm 3, the time complexity of Algorithm 4 is linear (i.e., $\mathcal{O}(n)$), with respect to the workload of the incoming DAG (i.e., W_b in the algorithm). In addition, implementation efforts can be conducted to further speed up the algorithm, by iterating execution blocks instead of time. However, we present the algorithm by iterating time for simplicity and better understanding.

Example 1. *Figure 8(a) provides an example illustrating the P-WDM accumulation of two DAGs (τ_a and τ_b) running on 6 cores. In this example, both DAGs are assigned with 4 cores and start at $t = 0$. The example adds the P-WDM of τ_b on τ_a, i.e., $WD_a^4 \oplus WD_b^4$.*

This example first illustrates the working mechanism of the $\oplus$ operator. As shown in the figure, the workload of τ_b can be added on the top of τ_a from $t = 0$ to $t = 2$, as the number of cores needed by both DAGs does not exceed the total number of cores. However, from $t = 2$ to $t = 6$, the system cannot provide enough cores to allocate τ_b's workload, which requires 4 cores. Under this case, τ_b is delayed until $t = 6$. After $t = 6$, the number of cores τ_b required are satisfied and the rest workload of τ_b is added on τ_a to form a new P-WDM.

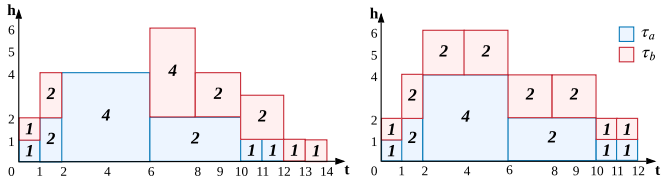
More importantly, this example explains the application of the analysis for multi-DAG systems. In this example, when the number of free cores is less than the cores τ_b needs, τ_b will be delayed instead of utilising fewer cores to "execute", which may seem counter-intuitive. However, by allowing this delay we can avoid the "offset" effect for τ_b, where the start of non-critical nodes in τ_b will not be delayed due to other DAGs in the system. By guaranteeing that each DAG can run on a fixed number of cores during the entire execution, we transform the multi-DAG analysis problem into the analysis of a set of single DAGs, with an additional delay to reflect the inter-task interference. This is because the analysis ensures: (i) the number of cores assigned to a DAG is fixed during the entire execution of the DAG; and (ii) the start time of nodes in the DAG is not prolonged due to other DAGs in the system.

Based on this model, the inter-task interference can be bounded by the delay a task can incur when being added to the current P-WDM of the system, which reflects the competition for cores between the DAG tasks. The intra-task interference can then be computed directly by the proposed analysis, which takes the number of cores being assigned as the input. By summing these two factors, the response time of a task in a multi-DAG system can be obtained.

We note that the P-WDM and the $\oplus$ operator are developed to facilitate the application of the single-DAG analysis on multi-DAG systems. During execution, the DAG can run as long as there exist idle cores, within the number of cores assigned to the DAG. That is, in Figure 8(a) the execution block with a height of four can execute along with τ_a in the real-world, without incurring any delay. However, in the viewpoint of P-WDM, τ_b incurs a delay of four units of time so that the proposed analysis can be effectively applied. The complete analysis and proof of correctness are presented later in Section VII-D.

Example 2. *Figure 8(b) provides another scenario of the workload accumulation of τ_a (in blue) and τ_b (in red). In this case, τ_a is assigned with 4 cores but τ_b can only execute on 2 cores. The example adds the P-WDM of τ_b on τ_a, i.e., $WD_a^4 \oplus WD_b^2$.*

In this example, the number of cores required by τ_b can be satisfied during its entire "execution" so that all workload in WM_b^2 can be added on WM_a^4 directly without any delay. Compared with Example 1, two observations can be obtained. First, although with fewer cores



(a) when τ_b is assigned with 4 cores (b) when τ_b is assigned with 2 cores

Figure 8: Then P-WDM accumulation.

assigned, the workload of τ_b in this example can be allocated with 2 units of time shorter, i.e., τ_b in Example 2 is a shorter "finish time". Second, with fewer cores being assigned, the "execution" duration of τ_b is two units of time longer that in Example 1.

This example highlights the trade-off between the delay due to core competition and the execution duration (i.e., the time spent for computation), which is controlled by the number of cores being assigned to the task. Although the execution duration of τ_b becomes longer with fewer cores assigned, τ_b can finish earlier in this case as it does not incur any delay from other DAGs in the system. This trade-off motivates the proposed core assignment method, which aims to find a balanced solution between the number of cores that a DAG can use and the delay it incurs due to core competition with other DAGs in the system, so that the overall system schedulability can be improved.

D. Response time analysis and core assignment based on P-WDM

As illustrated in Example 1, the response time of a DAG job $J_{x,j}$ can be bounded by Equation 17, denoted as $R^{\diamond}_{x,j}$. Notation $R_{x,j}$ gives the makespan of $J_{x,j}$ by the single-DAG analysis and $I_{x,j}$ gives the inter-task interference by Algorithm 4. Note, as different jobs of τ_x can be assigned with different number of cores, we use $R_{x,j}$ instead of R_x to differentiate each job instance. The notion of the number of cores that $J_{x,j}$ can execute on is encoded in Equation 2 and the construction (as well as the accumulation) of P-WDMs.

$$R^{\diamond}_{x,j} = R_{x,j} + I_{x,j} \tag{17}$$

Theorem 2. *The response time of $J_{x,j}$ is bounded by Equation 17.*

Proof. A DAG can incur both intra-task and inter-task interference in a Multi-DAG system [1], [16]. Below we prove $R^{\diamond}_{x,j}$ can upper bounds both types of interference.

For the inter-task interference, it is bounded by examining the P-WDM of the system and the incoming DAG based on P-WDM, where the DAG is delayed as long as the system cannot provide the number of core assigned (see Figure 8(a)). This upper bound is safe because the P-WDM is constructed based on the WCET of nodes, which depicts the worst-case workload of a DAG and the system. As with He[1], we focus on systems where no dependency exists between DAGs (see Section II-B). Therefore, a reduction in execution time of a DAG can only lead to less competition for cores, i.e., less inter-task interference.

As for the intra-task interference, it is effectively bounded by Equation 2 for two reasons. First, the analysis has been proved to bound the intra-task interference of a single DAG (see Lemmas 2-7 in Section V). Second, in a multi-DAG system, a reduction in execution time of DAG τ_a will not jeopardise the analysis of intra-task interference of an incoming DAG τ_b. With the work-conserving schedule applied, when nodes in τ_a execute less than their WCETs, nodes in τ_b can also start (and finish) earlier, if possible on the number

Algorithm 5: Offline core assignment for multi-DAGs

Inputs : Γ, m
Outputs : $nop_{x,j}, \forall J_{x,j} \in \Gamma^H$
Initialise : $WD^m = \varnothing$

1 get and sort Γ^H by (i) earliest release time first and (ii) highest priority first;
2 **for** each $J_{x,j} \in \Gamma^H$ **do**
3 $\check{R}^{\diamond}_{x,j} = \infty$;
4 **for** $i = m...1$ **do**
5 get $WD^i_{x,j}$ by Algorithm 3;
6 $(WD^m, I_{x,j}) = WD^m \oplus WD^i_{x,j}$;
7 compute $R_{x,j}$ by Equation 2 with i cores;
8 $R^{\diamond}_{x,j} = R_{x,j} + I_{x,j}$;
9 **if** $R^{\diamond}_{x,j} < \check{R}^{\diamond}_{x,j}$ **then**
10 $\check{R}^{\diamond}_{x,j} = R^{\diamond}_{x,j}$;
11 **else**
12 reset WD^m;
13 **end**
14 **end**
15 **if** $\check{R}^{\diamond}_{x,j} > d_{x,j}$ **then**
16 **return** *infeasible*;
17 **end**
18 **end**
19 **return** $\{nop_{x,j}, \forall J_{x,j} \in \Gamma^H\}$

of cores being assigned. As the single-DAG analysis is proved to be sustainable (see Section V-C), this will not lead to a $R_{x,j}$ higher than the worst case. For the same reason, allowing τ_b to execute earlier when there is fewer cores available than the number of cores assigned to τ_b does not jeopardise the bound. □

To this end, the offline core assignment algorithm for multi-DAGs can be constructed, as shown in Algorithm 5. The algorithm takes a set of sporadic DAG tasks Γ and the number of cores of the system m as the input, and produces a schedulable configuration with the minimal response time (if found) for each job in one hyper-period. A configuration of a job $J_{x,j}$ specifies the number of cores $J_{x,j}$ can execute on, denoted as $nop_{x,j}$.

Given the inputs, the algorithm first gets all jobs in one hyper-period (denoted as Γ^H) from Γ, and sorts the jobs by their release time in a non-decreasing order (line 1). If two DAGs are released at the same time, they will be sorted by priority non-increasingly. Then, starting from m, the algorithm searches for the configuration that leads to the minimal response time of the job, based on (i) the P-WDM of the current system and the job (lines 5-6, Algorithm 4) and (ii) the multi-DAG analysis in Equation 17 (lines 7-8). If a schedulable configuration is found, the algorithm assigns the current configuration to the DAG job, and the search continues for the next job (line 9-10). If a job is found unschedulable with any configuration, the algorithm returns immediately with no schedulable solution being found (line 15-17). A system is deemed schedulable if each job in one hyper-period is found schedulable with a given configuration.

Assuming linear complexity of the analysis, the time complexity of this algorithm is $\mathcal{O}(n^4)$, depending on the number of DAG jobs in one hyper-period $|\Gamma^H|$ and the number of cores in the system (i.e., m). In the worst case, $(|\Gamma^H| \times m)$ iterations are required to test the schedulability of each job under each configuration based on the P-WDM model. In each iteration, Algorithms 1 to 4 will be invoked to compute the priority of the DAG and its P-WDM, leading to a time complexity of $(|\Gamma^H| \times m \times n^2) \approx n^4$, where n^2 is the highest time complexity among the invoked algorithms, i.e., Algorithm 2. However, as described, the constructed multi-DAG scheduling method is performed offline to compute the number of cores each DAG can use. During run-time, the scheduler takes this configuration and executes DAGs based on their priorities, which has a run-time cost similar to that of a fixed priority scheduler with a global scheme [35].

In addition, it is worth noting that proposed single and multi-DAG methods are developed for homogeneous systems. As for heterogeneous platforms, the key challenge is raised by the different speed and types (e.g., FPGA or GPU) of the processing units, which can lead to fundamental changes of the DAG structure (e.g., the critical path), and subsequently, the CPC and WDM of a DAG when executing on different processing units [36]. In addition, the constraint that certain nodes can only be executed on a specific type of processing units further exacerbates the complexity of this problem [37]. To tackle this challenge, the constructed methods need to take into account the knowledge of which cores the DAG will execute on. Then, an analyse will be required to compute the impact of cores to the DAG structure and the constructed models before making scheduling and allocation decisions. Extending the constructed methods to heterogeneous system will be addressed in our future work.

VIII. Evaluations

The objectives of this evaluation are multifold: (i) to demonstrate the analysis (*rta-cpf* in Section V-B and *rta-cpf-eo* in Section V-D) and node execution ordering (Section VI) improve the worst-case makespan (using the classic bound as reference) of a DAG; (ii) to establish the conditions in which the proposed methods lead to an improved makespan of a DAG; (iii) to demonstrate the proposed execution order reduces makespan and the proposed single-DAG analysis tightens the worst-case bounds; and (iv) to demonstrate the offline core assignment has a higher system schedulability than existing methods on multi-DAG systems due to tighter bounds on the inter-task interference. The proposed node execution order (denoted as *EO*), analysis and the multi-DAG methods are compared with methods in He[1] (denoted as He2019 hereafter) in which a node priority assignment is proposed alongside the analysis for both single and multi-DAGs. Objectives (i) to (iii) show the contributions from Zhao[9]. Objective (iv) represents the most significant contribution on multi-DAGs made in this paper.

The experiment is evaluated through randomly generated DAGs. Each DAG task is generated as follows: the generator starts from a source node, and then generates nodes layer by layer. The maximum depth (the number of layers) is randomly chosen from 5 to 8. The number of generated nodes in each layer is uniformly distributed from 2 to the parallelism parameter, p. Open-ended nodes randomly add connections with a probability of $p_c = 0.5$ to join the other nodes in the previous layer. Then, all terminal nodes are connected to a sink node. The source and sink nodes serve the purpose of organising the node graph, they both have a execution time of one unit. Finally the execution times are randomly assigned to nodes given a total workload of W[1].

A. Evaluation of the worst-case makespan

This section evaluates the worst-case makespan produced by the constructed methods and the state-of-the-art [1], by varying the

[1]The evaluation implementation can be accessed at https://github.com/automaticdai/research-dag-scheduling-analysis.

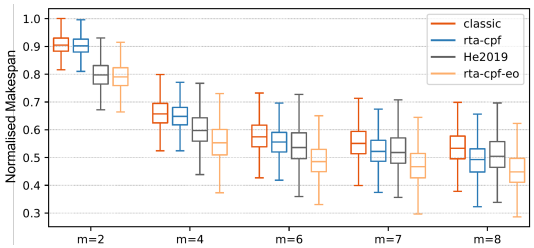


Figure 9: Normalised DAG worst-case makespan using analytical methods with varied number of cores (m).

number of cores (m). For each configuration (task and system setting), 1,000 trials are applied to the compared methods. Each trial generates one DAG task randomly. The makespan is normalised by the largest value observed and is used as the indicator (i.e., the y-axis).

Observation: Figure 9 presents the worst-case makespan of the existing and the proposed methods with a varied number of cores, on DAGs generated with $p = 8$. With $m \leq 4$, the *rta-cpf* provides similar results to the classic bound, i.e., most of its results are upper bounded by the classic bound. This is because with a small number of cores, the parallelism degree of the DAG is limited so that each non-critical node has a high worst-case finish time (see Equation 3). This leads to a low α_i bound (as well as a high β_i bound) for each provider and hence a longer worst-case makespan approximation. With m further increased, the *rta-cpf* becomes effective and outperforms the classic bound, e.g., by 15.7% and 16.2% on average when $m = 7$ and $m = 8$, respectively. In this case, more workload can execute in parallel with the critical path, i.e., an increase in α_i and a decrease in β_i. Thus, the *rta-cpf* leads to tighter results by explicitly accounting for such workload, resulting in a safe reduction in interference on the critical path.

Similar observations are also obtained in the comparison of *rta-cpf-eo* and He2019, where *rta-cpf-eo* provides shorter worst-case makespan approximations with $m \geq 4$, e.g., by up to 11.1% and 12.0% with $m = 7$ and $m = 8$ respectively. We note that the node execution order in both methods can also affect the analytical worst-case bounds. In Section VIII-C, we compare the scheduling and analysing methods separately. Furthermore, we observe that with $m = 7$, *rta-cpf* (with random execution order) provides similar results with He2019, and outperforms He2019 with $m = 8$.

Summary: These experiments provide an overall comparison of the proposed methods and the existing ones. The observations demonstrate that the our methods in general can achieve lower DAG makespan with varying number of cores. However, it is not straightforward to understand how different DAG properties would impact the worst-case makespan. In the next section we evaluate the sensitivity of our methods to different DAG properties.

B. Sensitivity of DAG properties on the evaluated methods

This experiment shows how the evaluated analysis is sensitive to certain DAG characteristics. That is to say, by controlling the parameters of the DAGs and evaluating the makespan in normalised values, it can be seen by how much the performance of the analysis changes. This would otherwise not be distinguishable through worst-case makespan or schedulability analysis. Specifically, we consider the following parameters in this experiment (with the number of cores fixed): (i) DAG parallelism (the maximum possible width when generating the randomised DAG), p; and (ii) DAG critical path ratio to the total workload, $\%L$, where $\%L = L/W$.

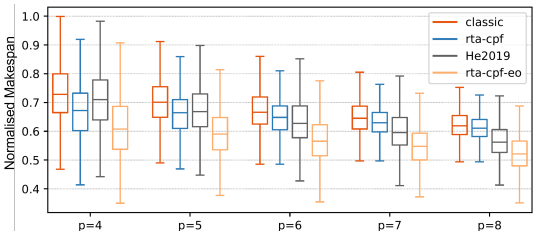


Figure 10: Sensitivity of parallelism parameter (p) when $m = 4$.

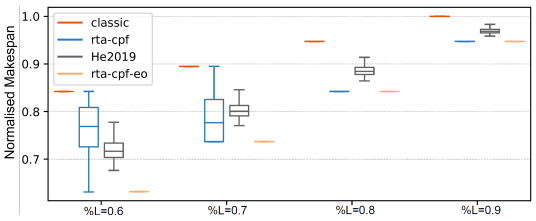


Figure 11: Sensitivity of critical path ratio ($\%L$) when $m = 2$, $p = 8$.

Observation: Figure 10 shows the worst-case makespan of the proposed methods with varied values of the parallelism parameter (with $m = 4$). Note, DAGs in this experiment are randomly generated as with Figure 9 so that $\%L$ is not controlled explicitly. As shown in the figure, given a fixed number of cores, *rta-cpf* outperforms the classic bound in general. However, with the increase of p, the difference in performance of both methods becomes less significant. The intuition behind this observation is, with an increased number of concurrent nodes, the interference set of each node also increases (see Equation 4), which then results in an increased worst-case finish time. This undermines the effectiveness of *rta-cpf*, which accounts for α_i and β_i based on worst-case finish time.

However, *rta-cpf-eo* demonstrates a strong performance and its effectiveness is not affected by the change on p, which consistently outperforms other methods in all system settings. This is because with an explicit execution order, the increase of concurrent nodes cannot impose a significant effect on the finish time of nodes, in which high priority nodes can execute immediately without any delay (see Equation 12). Therefore, *rta-cpf-eo* with parallelism DAGs can still account for the actual interfering workload effectively, and provide the lowest worst-case makespan.

Observations: Figure 11 evaluates the impact of the length of the critical path on the effectiveness of the proposed methods, with $m = 2$. The critical path is varied in a range from 60% to 90% of total workload of generated DAGs. In this experiment, the proposed analysis demonstrates the most pronounced performance compared to the existing methods.

For the proposed methods, the worst-case makespan of *rta-cpf* varies with a small number of $\%L$, due to the varied internal structure of the generated DAGs (e.g., $\%L = 0.6$). However, with a further increase of both $\%L$, *rta-cpf* provides a constant makespan, as all non-critical workload can execute in parallel with the critical path. In this case, the makespan directly equals the length of the critical path. Similar observations are also obtained for *rta-cpf-eo*, which provides a constant makespan (i.e., the length of critical path) under all experimental settings. Note, with further increases of $\%L$, He2019 is completely dominated by *rta-cpf* (based on evaluations but not presented due to page limitation).

Summary: Based on the above experiments, the proposed methods outperform the classic method and the state-of-the-art in a general

Table IV: Percentage of improvement in advantage cases w.r.t. node ordering policy

	EO $\succ$ He2019							EO $\prec$ He2019						
	m=2	m=3	m=4	m=5	m=6	m=7	m=8	m=2	m=3	m=4	m=5	m=6	m=7	m=8
avg.	7.89	8.05	7.21	6.77	6.18	5.72	5.41	6.47	5.92	4.53	3.24	2.52	1.64	1.65
max.	30.63	36.18	33.39	34.17	30.65	27.75	25.27	30.68	27.19	23.83	21.59	24.09	16.76	19.26
min.	0.05	0.02	0.02	0.02	0.02	0.02	0.03	0.01	0.04	0.02	0.03	0.03	0.02	0.03

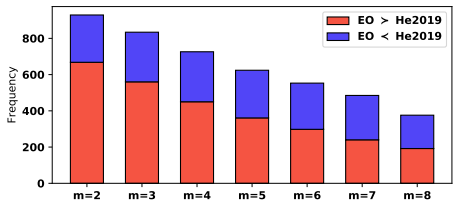


Figure 12: Proposed priority ordering v.s. the ordering in He[1], grouped by the number of cores (m), with $p = 8$. In the legend, "$\succ$" indicates outperforming and "$\prec$" means the vice versa.

Table V: Advantage cases and scientific significance in node ordering – both EO and He2019 are implemented in (α, β) analysis.

m	Dataset	# of data	Magnitude
2	EO $\succ$ He2019	668	medium
	He2019 $\succ$ EO	261	medium
4	EO $\succ$ He2019	450	medium
	He2019 $\succ$ EO	276	small
6	EO $\succ$ He2019	298	small
	He2019 $\succ$ EO	255	negligible
8	EO $\succ$ He2019	192	small
	He2019 $\succ$ EO	184	negligible

case. In addition, we observed that each of the tested parameters m, p, $\%L$ has an impact on the performance of the proposed methods. For *rta-cpf*, it is sensitive to the relation between m, p, in which a low m or a high p undermines the effectiveness of the method. Both factors have a direct impact on the finish time of all non-critical nodes. In addition, $\%L$ can also significantly affects the performance of *rta-cpf*, in which a long critical path generally leads to more accurate makespan approximations. Similar with *rta-cpf*, *rta-cpf-eo* demonstrates better performance with the increase of $\%L$. However, due to its explicit execution order, *rta-cpf-eo* shows much stronger performance than *rta-cpf* and is not affected by parameter p.

C. Effectiveness of the proposed schedule and analysis

In this experiment, the proposed priority assignment is compared against the assignment in He[1] in terms of the worst-case DAG makespan. Overall there are 1000 random tasksets generated under each configuration. Two metrics are compared in this evaluation: (a) the percentage of times that the proposed *rta-cpf-eo* analysis is better than the compared method, and (b) the reduction in the normalised makespan within the improved cases.

Observation: Figure 12 reports the comparison of the proposed ordering method and the method in He[1], with a varied number of cores. The term "frequency" indicates the number of cases that the proposed schedule has a shorter (in red) or longer (in blue) makespan than He2019. For fairness, the proposed worst-case makespan analysis for explicit order (Section V-D) is applied for both ordering, so the differences in performance all comes from the ordering policies.

From the results, the proposed method outperforms He2019 with a higher frequency in general, especially with a small number of cores, e.g., around the frequency of 600 with $m = 2$ and $m = 3$. With the increase of m, the difference in frequency of the methods gradually decreases, and becomes difficult to distinguish with $m = 7, 8$. In these cases, most nodes can execute in parallel so that different execution order have less impact on the final makespan. For the same reason, both methods have an increasing frequency to produce the same makepsan with the increase of m. This explains the decreasing total frequency in the figure, which only reports the cases where one method outperforms another.

Table IV presents detailed comparison of both methods in their advantage cases, in terms of the percentage of improvements. For EO $\succ$ He2019 (i.e., proposed schedule outperforms He2019), we observe an average improvement (in terms of worst-case makespan) higher than 5.4% (up to 7.89%) in all cases. For cases with EO $\prec$ He2019 (i.e., He2019 performs better), the improvement is consistently lower than the corresponding case with EO $\succ$ He2019.

Table V reports the number of advantage cases and the scientific significance of the improvements, in both EO $\succ$ He2019 and EO $\prec$ He2019. The magnitude in Table V is a categorical value in (negligible effect, small effect, medium effect and large effect) to reflect the scientific significance [38]. In other words, the scientific significance informs whether any difference is more than random chance and the size of the difference. The column *# of data* illustrates the number of times one approach has a lower makespan than the other. In all cases our approach outperforms the state of the art. The *Magnitude* gives further evidence of the benefits of our approach, e.g., $m = 4$ the effect size when EO outperforms He2019 is medium versus small for He2019 outperforming EO, and for $m = 8$ it is small versus negligible even though *# of data* have similar values.

Summary. The above experiments show the benefits of the proposed single-DAG scheduling method. A similar comparison of our analysis and the one in He[1] by applying the same execution order is also conducted with consistent results being observed (not presented due to page limitation). Therefore, we conclude that the proposed scheduling and analysing are effective, and outperform the state-of-art techniques in the general case.

D. Evaluation of the schedulability of multi-DAG systems

The objective of this section is to demonstrate the proposed multi-DAG core assignment (denoted as mDAG-CA) achieves higher system schedulability than methods in He[1] and the non work-conversing approach in Zhao[10] (denoted as Zhao2020), and achieves lower response time compared to a variant of mDAG-CA (denoted as the Baseline). The baseline method shares the same process with mDAG-CA but returns immediately if a schedulable configuration is found instead of minimising the response time of the DAG. Two experiments are conducted. The first evaluates the resulting system schedulability of the competing methods. Then, the second experiment reveals the key reason that mDAG-CA has a higher schedulability by examining the *intra-task* and *inter-task* interference respectively of the evaluated methods.

The experiment setup is as follows unless specified otherwise: the number of cores is set to $m = 8$. The total utilisation of the system

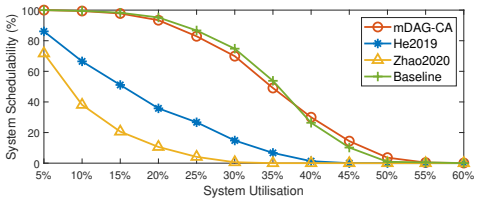


Figure 13: System schedulability comparison with scaled U.

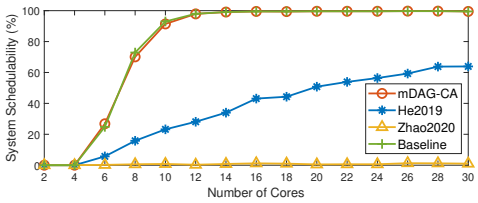


Figure 14: System schedulability comparison with scaled m.

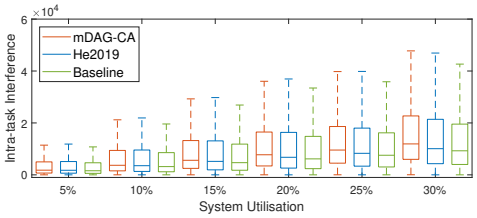


Figure 15: The *intra-task* interference comparison with scaled U.

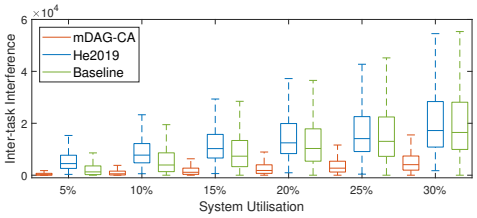


Figure 16: The *inter-task* interference comparison with scaled U.

ranges from 5% to 100%, with a step size of 5%. A taskset has 5 DAG tasks, and each DAG is generated randomly in the same way as introduced earlier. We use the Dirichlet-Rescale algorithm in Griffin[39] to generate and ensure the uniformity of the utilisation of DAGs in a taskset as well as the execution time of nodes in each DAG based on its workload $W_i = U_i/T_i$. The periods of DAG tasks are generated randomly in a uniform distribution from all periods that lead to a hyper-period of $1440ms$, and deadlines are equal to periods. The priorities are assigned to DAGs based on the deadline-monotonic policy. The system schedulability of He2019 is computed by its multi-DAG analysis under a non-preemptive scheduling scheme. The schedulability of Zhao2020 is computed by its multi-DAG analysis for the non work-conserving schedule.

Experiment One: Figures 13 and 14 present the system schedulability of mDAG-CA, He2019, Zhao2020 and the Baseline by scaling the system utilisation U and the number of cores m, respectively.

Observation: As shown in Figure 13, the mDAG-CA demonstrates a higher schedulability compared to H2019 and Zhao2020, in which the mDAG-CA can schedule more than 90% systems with $U = 20\%$ while He2019 and Zhao2020 can only schedule 40% and 10% of systems, respectively. This observation is expected because based on the P-WDM, mDAG-CA effectively reduces the pessimism in the inter-task analysis. For He2019, it outperforms Zhao2020 but demonstrates a quick fall off when U is increasing. As discussed at the beginning of Section VII, this is due to the pessimism in its inter-task analysis, which does not take DAG parallelism into account. For Zhao2020, it is not surprising that it has the lowest schedulability. This is due to its non-work-conserving approach, where the next ready DAG can execute only if the current one is finished. In addition, the Baseline method demonstrates a similar schedulability with the mDAG-CA. This is expected as both methods share a similar process for producing schedulable solutions. The key difference between mDAG-CA and the Baseline will be revealed in experiment two.

The same observations are also obtained in Figure 14 by varying the number of cores (m), where the mDAG-CA is similar with the Baseline and is consistently better than He2019 and Zhao2020. Note that in Figure 14, the workload of the DAGs is fixed (which is equivalent to the workload of systems with $U = 30\%$ and $m = 8$) so that we only change one parameter (i.e., m) at a time. With $m \in [1, 4]$, all methods cannot schedule any system because the workload is too high for the given system configuration. With $m > 4$, the mDAG-CA and the Baseline demonstrate a strong schedulability, where they can schedule more than 90% of systems when $m = 10$. In contrast, He2019 has a much slower increasing trend in schedulability because of the pessimism in its inter-task interference, where it can only schedule 20% systems with $m = 10$. As for Zhao2020, it cannot schedule any system in this case as the increase in m does not bring any effect to its non work-conserving schedule.

Experiment Two: To provide evidence that the mDAG-CA achieves high schedulability due to its tighter bound on the inter-task interference in Figures 13 and 14, the second experiment presents the intra-task and inter-task interference of tasks under each evaluated method. The experimental results are presented in Figures 15 to 18. In the figures, each trial reports the intra-task (or inter-task) interference of 1000 schedulable systems by mDAG-CA, He2019 and the Baseline with the given system configuration. The outliers are removed for the sake of readability.

Observation: Figures 15 and 16 present the intra-task and inter-task interference of the mDAG-CA, He2019 and the Baseline with scaled system utilisation U. In Figure 15 we observe that the intra-task interference of all methods increases along with the increase of U, in which mDAG-CA is slightly higher. This is because mDAG-CA can assign a number of cores less than m to a DAG job based on the P-WDM, resulting in higher intra-task interference but a lower inter-task interference to reduce the total response time. The key reason that mDAG-CA can achieve higher schedulability is highlighted in Figure 16, where it demonstrates a much lower inter-task interference than He2019 and the Baseline method. The inter-task interference in He2019 is computed based on the sum of workload of all high priority DAGs released during the release of τ_x (i.e., $\sum_{\tau_j \in hp(x)} \left\lceil \frac{R_x}{T_j} \right\rceil \cdot W_j$ in Equation 14). With an increasing utilisation, this interfering workload becomes significant so that He2019 has a large bound on the interference, and subsequently, low schedulability in Figure 13. In addition, although the Baseline method shares a similar process as the mDAG-CA, it does not focus on minimising the response time. Therefore, even if a similar schedulability is achieved, it demonstrates a higher inter-task interference compared to the mDAG-CA, especially when the system workload is high (e.g., when $U = 30\%$). In contrast, the mDAG-CA aims to minimise the response time of each DAG, which leads to a tighter bound on the inter-task interference and high system schedulability.

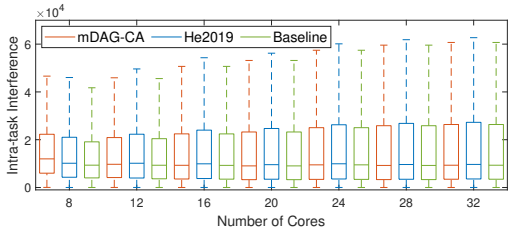


Figure 17: The *intra-task* interference comparison with scaled m.

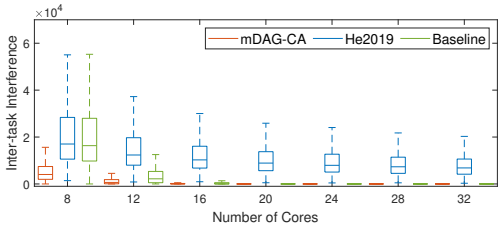


Figure 18: The *inter-task* interference comparison with scaled m.

Figures 17 and 18 present the intra-task and inter-task interference of the mDAG-CA, He2019 and Baseline with scaled number of cores m. We first observed that with the increase of m, DAGs with a higher workload become schedulable so that intra-task interference of all methods are slightly increased. In addition, with a large number of m (e.g. $m = 12$), the intra-task interference of mDAG-CA and Baseline becomes slightly lower than that of He2019 (Figure 17). This observation is in line with previous sections that with a large number of cores, the proposed single-DAG analysis is less pessimistic by explicitly computing the parallel workload which does not impose a delay. As for the inter-task interference (Figure 18), the mDAG-CA is able to provide a very tight bound, where it is close to zero when $m \geq 16$. This highlights the key advantage of the proposed method, which bounds the inter-task interference by a parallelism-aware manner. Notably, the Baseline method produces a low inter-task interference in this experiment. This is because the increase of m naturally leads to a lower inter-task interference, which can be precisely computed by the Baseline method (as well as the mDAG-CA) based on the P-WDM of each DAG and the system. Therefore, with $m \geq 16$, the Baseline method significantly reduces the pessimism of the analysis and provides tight analytical bounds. However, He2019 can still impose certain inter-task interference due to lacking the knowledge of DAG parallelism, even if most of the DAGs can execute in parallel given a large number of cores.

Summary: In these experiments, we show that the mDAG-CA has a higher system schedulability than the competing method. Further, an in-depth examination of the intra-task and inter-task interference provides evidence that the high schedulability of the proposed method is achieved by its tighter bound on the inter-task interference.

Computation cost: Table VI reports the average computation cost of the complete multi-DAG scheduling process, i.e. Algorithms 1 to Algorithm 5. The measurement is performed in a simulation environment on a Intel(R) Core(TM) i7-6700K CPU. The setup is identical to Figure 17 with $m = 8$, and each algorithm are executed and measured for 10,000 times. From the table we observe Algorithms 1 to 4 have a relatively low computation cost, i.e. less than 5 milliseconds. However, for the complete multi-DAG scheduling process (Algorithm 5), its computation cost is quite high, i.e., above 2 seconds. This is because the measurement also takes into account the cost for running the proposed analysis (Section V). However, as described in Section VII, the construct multi-DAG scheduling approach is an offline method. During run-time, the scheduler only executes DAGs based on the pre-determined configuration, which has a similar cost to the fixed priority scheduler with a global scheme.

Table VI: The average computation cost of the complete multi-DAG scheduling process (i.e. Algorithms 1 to 5) in milliseconds.

Algorithm	Computation Cost	Standard Deviation	Time Complexity
Alg. 1	1.934 ms	1.234	$\mathcal{O}(n)$
Alg. 2	3.476 ms	2.075	$\mathcal{O}(n^2)$
Alg. 3	4.870 ms	12.173	$\mathcal{O}(n)$
Alg. 4	0.095 ms	0.014	$\mathcal{O}(n)$
Alg. 5	2188.5 ms	202.153	$\mathcal{O}(n^4)$

IX. Concluding Remarks

In this paper, a CPC model is constructed for a DAG to provide an in-depth understanding of inter-node dependency and parallelism. Based on CPC, a response time analysis is developed for a single recurrent DAG that provides tighter bounds than existing analysis for (i) any scheduling method that prioritises the critical path, and (ii) scheduling methods with explicit execution order known a priori. A rule-based scheduling method is proposed which maximises node parallelism to reduce makespan.

To support scheduling and analysing of multi-DAG systems, we develop the P-WMD to understand the workload distribution of a DAG under a given number of cores. Based on this model, an offline core assignment method is constructed to address the problem of applying the single-DAG analysis and to facilitate the scheduling on multi-DAG systems. We demonstrate that the proposed scheduling and analysing methods outperform existing techniques, and the proposed core assignment can achieve higher schedulability.

References

[1] Q. He, X. Jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2283–2295, 2019.

[2] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Real-Time Systems Symposium*, 2012, pp. 63–72.

[3] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate DAGs from multi-rate task sets," in *Real-Time and Embedded Technology and Applications Symposium*, 2020, pp. 226–238.

[4] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *International Conference on Embedded and Real-Time Computing Systems and Applications*, 2016, pp. 159–169.

[5] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "Rosch: real-time scheduling framework for ROS," in *International Conference on Embedded and Real-Time Computing Systems and Applications*, 2018, pp. 52–58.

[6] Y. Suzuki, T. Azumi, N. Nobuhiko, and S. Kato, "HLBS: Heterogeneous laxity-based scheduling algorithm for DAG-based real-time computing," in *International Conference on Cyber-Physical Systems, Networks, and Applications*, 2016, pp. 83–88.

[7] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 301–310.

[8] S. E. Saidi, N. Pernet, and Y. Sorel, "Automatic parallelization of multi-rate fmi-based co-simulation on multi-core," in *Symposium on Theory of Modeling and Simulation* , 2017, p. Article No. 5.

[9] A. Vincentelli, P. Giusto, C. Pinello, W. Zheng, and M. Natale, "Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems," in *Real Time and Embedded Technology and Applications Symposium*, 2007, pp. 293–302.

[10] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang, "DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency," in *IEEE Real-Time Systems Symposium*, 2020, pp. 128–140.
[11] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of dags," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3242–3252, 2014.
[12] N. Ueter, G. von der Brüggen, J.-J. Chen, J. Li, and K. Agrawal, "Reservation-based federated scheduling for parallel real-time tasks," in *2018 IEEE Real-Time Systems Symposium*, 2018, pp. 482–494.
[13] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *International Conference on Real-Time Networks and Systems*, 2017, pp. 28–37.
[14] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *IEEE Real-Time Systems Symposium*. IEEE, 2018, pp. 421–433.
[15] N. C. Audsley, A. Burns, and A. J. Wellings, "Deadline monotonic scheduling theory and application," *Control Engineering Practice*, vol. 1, no. 1, pp. 71–78, 1993.
[16] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *Euromicro Conference on Real-Time Systems*, 2015, pp. 211–221.
[17] R. L. Graham, "Bounds on multiprocessing timing anomalies," *Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.
[18] S. Chang, X. Zhao, Z. Liu, and Q. Deng, "Real-time scheduling and analysis of parallel tasks on heterogeneous multi-cores," *Journal of Systems Architecture*, vol. 105, p. 101704, 2020.
[19] F. Guan, J. Qiao, and Y. Han, "DAG-fluid: A real-time scheduling algorithm for DAGs," *IEEE Transactions on Computers*, no. 01, pp. 1–1, 2020.
[20] P. Chen, W. Liu, X. Jiang, Q. He, and N. Guan, "Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5, pp. 1–19, 2019.
[21] H. Lin, M.-F. Li, C.-F. Jia, J.-N. Liu, and H. An, "Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems," *Journal of Computer Science and Technology*, vol. 34, no. 5, pp. 1096–1108, 2019.
[22] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.
[23] 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 *Euromicro Conference on Real-Time Systems*, 2014, pp. 85–96.
[24] S. Baruah, "The federated scheduling of systems of conditional sporadic DAG tasks," in *International Conference on Embedded Software*, 2015, pp. 1–10.
[25] N. Ueter, G. Von Der Brüggen, J.-J. Chen, J. Li, and K. Agrawal, "Reservation-based federated scheduling for parallel real-time tasks," in *2018 IEEE Real-Time Systems Symposium*. IEEE, 2018, pp. 482–494.
[26] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-federated scheduling of parallel real-time tasks on multiprocessors," in *Real-Time Systems Symposium*, 2017, pp. 80–91.
[27] T. Yang, Y. Tang, X. Jiang, Q. Deng, and N. Guan, "Semi-federated scheduling of mixed-criticality system for sporadic DAG tasks," in *International Symposium on Real-Time Distributed Computing*, 2019, pp. 163–170.
[28] M. Shariati, M. Naghibzadeh, and H. Noori, "Semi-federated scheduling of multiple periodic real-time dags of non-preemptable tasks," in *2018 8th International Conference on Computer and Knowledge Engineering*. IEEE, 2018, pp. 84–91.
[29] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, "Response time analysis of sporadic DAG tasks under partitioned scheduling," in *Symposium on Industrial Embedded Systems*, 2016, pp. 1–10.
[30] B. Bado, L. George, P. Courbin, and J. Goossens, "A semi-partitioned approach for parallel real-time scheduling," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, 2012, pp. 151–160.
[31] C. Maia, P. M. Yomsi, L. Nogueira, and L. M. Pinho, "Semi-partitioned scheduling of fork-join tasks using work-stealing," in *International Conference on Embedded and Ubiquitous Computing*. IEEE, 2015, pp. 25–34.
[32] M. Hatami, "Semi-partitioned scheduling hard real-time periodic dags in multicores," in *The Proceeding of First Work-in-Progress Session of 2018 CSI International Symposium on Real-Time and Embedded Systems and Technologies*, 2018, p. 9.
[33] A. Burns and S. K. Baruah, "Sustainability in real-time scheduling." *Journal of Computing Science and Engineering*, vol. 2, no. 1, pp. 74–97, 2008.
[34] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones, "Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions," in *Design, Automation & Test in Europe Conference & Exhibition*, 2016, pp. 1066–1071.
[35] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium*. IEEE, 2020, pp. 3–11.
[36] M. Han, N. Guan, J. Sun, Q. He, Q. Deng, and W. Liu, "Response time bounds for typed dag parallel tasks on heterogeneous multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2567–2581, 2019.
[37] M. Han, T. Zhang, Y. Lin, and Q. Deng, "Federated scheduling for typed dag tasks scheduling analysis on heterogeneous multi-cores," *Journal of Systems Architecture*, vol. 112, p. 101870, 2021.
[38] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[39] D. Griffin, I. Bate, and R. I. Davis, "Generating utilization vectors for the systematic evaluation of schedulability tests," in *IEEE Real-Time Systems Symposium*, 2020, pp. 76–88.

Shuai Zhao is a research associate in real-time system group, University of York, UK. He received a Ph.D. degree in computer science from the University of York in 2018, supervised by Prof. Andy Wellings. His research interests include scheduling algorithm, multiprocessor resource sharing, schedulability analysis, and safety-critical programming languages.

Xiaotian Dai is a research associate at the University of York, UK. He received a PhD degree from University of York in 2019 (with Best Thesis). He joined real-time systems group in 2015 as a PhD research student, supervised by Prof. Alan Burns. His PhD research involves cooperatively design of control system and real-time task scheduling for Cyber-Physical Systems. He serves as a reviewer and a Program Committee member for many top real-time and design automation conferences.

Iain Bate is a Professor in Dependable Real-Time Systems within the Department of Computer Science at the University of York. He is also the Deputy Head of the department. His research interests include scheduling and timing analysis, and the design and certification of Cyber Physical Systems. He has chaired a number of leading International Conferences and is a frequent member of Programme Committees. He was the Editor-in-Chief of the Microprocessors and Microsystems journal and then the Journal of Systems Architecture for 15 years. He leads a number of prestigious research projects in collaboration with industry.