DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency

Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, Wanli Chang
Department of Computer Science, University of York, UK
{shuai.zhao, xiaotian.dai, iain.bate, alan.burns, wanli.chang}@york.ac.uk

***Abstract*—With ever more complex functionalities being implemented in emerging real-time applications, multiprocessor systems are demanded for high performance, and directed acyclic graphs (DAGs) are used to model functional dependencies. In this work, we study a single periodic non-preemptive DAG running on a homogeneous multiprocessor platform, which is a common setup in many domains, such as automotive, robotics, and industrial automation. Aiming to reduce the makespan of the DAG and provide a tight yet safe bound, our contributions involve the exploitation of node-level parallelism and inter-node dependency, which are the two key factors of a DAG topology. First, we introduce a concurrent provider and consumer (CPC) model that precisely captures the above two factors, and can be recursively applied when parsing a DAG. Building upon CPC, we propose a novel scheduling method focused on reducing the makespan that orders the nodes in the following sequence: (i) the critical path, (ii) early predecessor paths of the critical path, and (iii) longer paths. Secondly, new response time analysis is presented, which provides a generic bound for any execution order of the non-critical nodes and a specific (tighter) bound for a fixed such order. Comprehensive evaluation demonstrates that our scheduling approach and analysis outperforms the state-of-the-art methods.**

I. Introduction

Driven by the demands of high performance and complex functionalities, multiprocessor systems are increasingly being employed in real-time applications. Directed Acyclic Graphs (DAGs) tasks are used to model functional dependencies [1].

Many existing works use a single recurrent event- or time-triggered DAG task to model the system [2], [3], [4], [5], [6], [7], [8]. For example, a complete automotive task chain from perception to control is described in [2] and converted to a single periodic DAG task.

In addition, to avoid migration and cache-related preemption overhead, a non-preemptive global scheduling scheme is often deployed [2], [9]. That is, the nodes of a DAG are scheduled globally on all cores and preemption is not allowed during the execution of a node [10].

Main contributions: In this work, we study a commonly seen setup, where a single periodic non-preemptive DAG runs on a homogeneous multiprocessor platform. By fully exploiting the node-level parallelism and inter-node dependency, which are the essence of the DAG topology, we reduce the makespan (i.e., the time interval between the starting and finishing of the DAG execution) and provide a tight yet safe bound on the makespan.

This paper has passed an Artifact Evaluation process.

The first principal contribution is to introduce a concurrent provider and consumer (CPC) model to precisely capture the two factors: parallelism and dependency. The CPC model describes the critical path (the longest execution sequence) in a DAG as a set of consecutive providers, each of which has a group of non-critical nodes (i.e., consumers) that can 1) execute concurrently with the provider and 2) delay the starting of the next provider. The intuition comes from that the non-critical nodes consume the computation resource (on other cores in parallel) provided when running a critical node. This model can be recursively applied to build nested CPC when parsing a DAG and serves as the foundation of the scheduling and analysis.

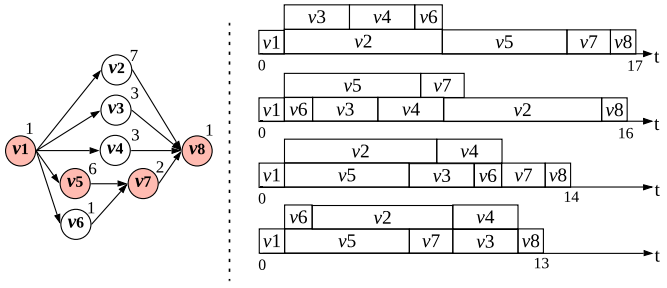
For the second contribution, a novel scheduling method for CPC is proposed that orders the nodes in the following sequence: (i) the critical path (i.e., providers), (ii) early predecessors paths of the critical path (i.e., consumers paths that would otherwise block the following providers), and (iii) longer paths (in a consumer group of a provider). Furthermore, we present new response time analysis that provides two provably safe bounds. One is a generic bound featuring critical-path-first execution with any execution order of the non-critical nodes, accounting for the workload that causes a delay. The other is a specific and tighter bound when the scheduler enforces a fixed order of the non-critical nodes. Comprehensive evaluation shows that our work outperforms the state-of-the-art methods.

Organisation: The rest of the paper is organised as follows: Section II presents the system and task model. Section III describes the state-of-the-art approaches in DAG scheduling and analysis with a motivational example. The proposed scheduling method is explained in Section IV, with the CPC model given in Section IV-A. Section V provides the new response time analysis. Section VI briefly discusses the extension to multiple DAGs. Finally, the evaluation is reported in Section VII before Section VIII makes the concluding remarks.

II. Task Model and Scheduling Preliminaries

A. Task model

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



(a) Example DAG (b) Execution scenarios

Figure 1: Makespan of a DAG task with example execution scenarios and the critical path highlighted.

any two nodes. Each node $v_{x,j} \in V_x$ represents a computation unit 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 the system has only one 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. Without loss of generality, we assume each DAG has one source and one sink node. Nodes that can execute concurrently with v_j are given by $\mathcal{C}(v_j) = \{v_k | v_k \notin (anc(v_j) \cup des(v_j)), \forall v_k \in V\}$ [11].

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 1(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 $\mathcal{C}(v_j) = \{v_2, v_3, v_4\}$. 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. Work-conserving schedule and analysis

The majority of the existing work on scheduling DAG tasks assumes a *work-conserving* scheduler [12]. 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 [13]. This analysis is later formalised in [12], [14] for DAG tasks, as given in Equation 1. Notation R_x denotes the response time of τ_x, m denotes the number of processors, $I_{x,y}$ gives the interference from a high priority DAG task τ_y, and $hp(x)$ gives all high priority tasks of τ_x.

$$R_x = L_x + \left\lceil \frac{1}{m}(W_x - L_x) \right\rceil + \sum_{\tau_y \in hp(x)} I_{x,y} \qquad (1)$$

In this analysis, the worst-case response time of a DAG task τ_x is upper bounded by the finish time of the critical path, with interference imposed by all non-critical nodes of τ_x itself and high priority DAG tasks, i.e., $I_{x,y}, \forall \tau_y \in hp(x)$. Details for bounding $I_{x,y}$ can be found in [12] and [14]. However, this analysis assumes a node v_j can be delayed by all the concurrent nodes [12], which is pessimistic for scheduling methods with an explicit execution order known a priori [11].

Figure 1(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 methods to reduce the run-time makespan and to tighten the analytical bounds of a single recurrent DAG task.

III. Related Work

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 [15], [16] or node-based [11], [17]. 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 [15]). 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 [18] and topological order of nodes [11]) or the *temporal* (execution time of nodes [17], [2], [19]) characteristics of the DAG. Below we describe two most recent node-based methods.

In [17], an anomaly-free non-preemptive scheduling method is proposed for a single periodic DAG, which always executes the ready node with the longest WCET to improve parallelism.

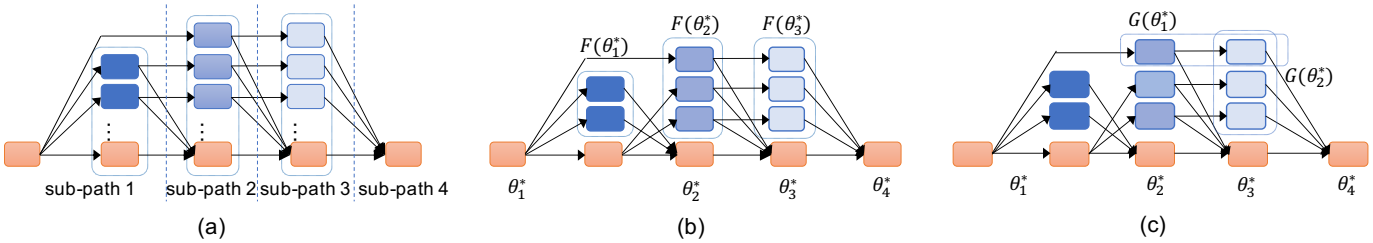


Figure 2: The CPC model of a DAG. The critical path is highlighted in orange and the non-critical nodes are in blue, with different colour gradation to indicate the earliness they can delay the critical path.

[17] prevents anomalies 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 1, 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 [11], a new response time analysis is presented, which dominates the traditional bound [13], [12] when an explicit node execution order is known a priori. That is, a node v_j can only incur 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 [11] 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, He et al. in [11] schedule 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. As illustrated in Section IV, this heuristic is not dependency-aware, which can reduce parallelism, and hence, lengthen the final critical path.

IV. DAG Scheduling: A Parallelism and Node Dependency Exploited Method

Equation 1 indicates that minimising the delay from non-critical nodes to the critical path (i.e., $\frac{1}{m}(W-L)$) effectively reduces makespan of the DAG. Achieving this requires the complete knowledge of the topology (i.e., the dependency and parallelism of each node) of a DAG so that the potential delay of the critical path can be identified. To support this the CPC model is proposed to fully exploit node dependency and parallelism (Section IV-A).

Based on the CPC model, a scheduling method is then presented to maximise node parallelism. 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. Firstly to always execute the critical path first (Section IV-B), and then two rules (Section IV-C) to maximise parallelism and to minimise the delay to the critical path.

The entire proposed approach has general applicability to DAGs with any topology (unlike, e.g., [14], which assumes nested fork-join DAGs only). It assumes a homogeneous architecture, however, it is not restricted by the number of processors. Table I summarises notations introduced by the proposed CPC model and scheduling method.

A. Concurrent provider and consumer model

The CPC model has two key stages. First, the critical path is divided into a set of consecutive sub-paths based on the potential delay it can incur (Figure 2(a)). Second, for each sub-path, the CPC model identifies the non-critical nodes that can 1) execute in parallel with the sub-path and 2) delay the start of the next sub-path, based on precedence constraints (Figure 2(b) and (c)).

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*, which is the length of the critical path. Note that non-critical nodes that utilise this capacity to execute cannot cause any delay to the critical path. The sub-paths in the critical path are termed *capacity providers* Θ^* and all non-critical nodes are *capacity consumers* Θ. For each provider $\theta_i^* \in \Theta^*$, it has 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}^* in the critical path.

Algorithm 1 presents a two-step process for constructing the CPC model of an input DAG $\mathcal{G}$ with its critical path λ^*. Starting from the head node in λ^*, capacity providers are formed by analysing node dependency between the critical path and non-critical nodes (Line 3-9). For a provider θ_i^*, its nodes should execute consecutively without delay from non-critical nodes in terms of dependency. That is, each node in θ_i^*, other than the head node (Line 5), only has one predecessor which is the previous node in θ_i^*, see Figure 2(b) with four capacity providers identified.

Then, for each $\theta_i^* \in \Theta^*$, its consumers $F(\theta_i^*)$ are identified as the nodes that 1) 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). Accordingly, nodes in $F(\theta_i^*)$ that finish later than θ_i^* will delay the start of θ_{i+1}^* (if it exists). In Figure 2(b), nodes in $F(\theta_1^*)$ can delay θ_2^* if they are finished later than θ_1^*. By doing so,

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

Inputs : $\{\mathcal{G} = (V, E)\}$
Outputs : $\Theta^*,\ F(\theta_i^*), G(\theta_i^*), \forall \theta_j^* \in \Theta^*$
Parameters: $\lambda^*, V^\neg = V \backslash \lambda^*$

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

the CPC model provides detailed knowledge of the potential delay caused by non-critical nodes on the critical path.

Furthermore, given an arbitrary DAG structure, a consumer $v_j \in F(\theta_i^*)$ can start earlier than, synchronous with, or later than the start of θ_i^*. For synchronous and late-released consumers, they will only utilise the capacity of θ_i^*. However, an early-released consumer can execute concurrently with certain previous providers, and therefore interfere with their consumers and impose an indirect delay to those providers. For a provider θ_i^*, $G(\theta_i^*)$ (in line 13) denotes the nodes that belong to the consumer groups of later providers, but which can execute in parallel (in terms of topology) with θ_i^*. In Figure 2(c), nodes in $G(\theta_1^*)$ and $G(\theta_2^*)$ belong to $F(\theta_2^*)$ and $F(\theta_3^*)$, but can execute in parallel with θ_1^* and θ_2^* respectively based on the precedence constraints.

With the CPC model, a DAG is transformed into a set of capacity providers and consumers, with a time complexity of $\mathcal{O}(|V|+|E|)$. The CPC model provides complete knowledge of both direct and indirect delays from non-critical nodes on the critical path. For each 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 $G(\theta_i^*)$.

Recall the DAG in Figure 1(a), its critical path forms three providers $\theta_1^* = \{v_1, v_5\}$, $\theta_2^* = \{v_7\}$ and $\theta_3^* = \{v_8\}$, and 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 $G(\theta_2^*) = G(\theta_3^*) = \varnothing$.

We now formally define the parallel and interfering workload of a capacity provider. 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 total workload of θ_i^*, $F(\theta_i^*)$ and $G(\theta_i^*)$. We formally define the terms *parallel* and *interfering* workload of a provider θ_i^*. Note, $W \leq \sum_{\theta_i^* \in \Theta} W_i$ as a consumer can be accounted for more than once if it can execute concurrently with multiple providers.

Table I: Notations introduced in the proposed CPC model and scheduling method.

Notation	Description
Θ^*/Θ	The set of capacity providers/consumers.
θ_i^*	A capacity provider with an index i.
p_j	The priority of a node v_j.
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^*.
$F(\theta_i^*)$	The consumer group of θ_i^*.
$G(\theta_i^*)$	Nodes in the consumer groups of later providers that can execute in parallel with θ_i^*.
$f(\cdot)$	The finish time of a given provider or a consumer node.
$l_j(\cdot)$	The length of the longest path that includes v_j in the set of input nodes.

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, 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 model, 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 "Critical Path First" execution (CPFE)

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: 1) a reduction on the length of the provider path, and 2) 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 $pre(v_j)$) [13], [12]. 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, $\alpha \geq \alpha'$. □

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

Theorem 1 leads to the first assignment rule that assigns critical nodes with the highest priority, in which p_j denotes the priority of node v_j. 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 in Figure 1, 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.

C. 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 [11], which avoids this delay by the heuristic of early interference node first.

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

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 $F(\theta_{i+1}^*)$. 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 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 1, , 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 [11], 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 belong 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 $F(\theta_i^*)$.

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. 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 $F(\theta_i^*)$ [11]. 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 1, 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.

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

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. The method starts from the outer-most CPC model ($CPC(\mathcal{G}, \lambda^*)$), and assigns all provider

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

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

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**

nodes with the highest priority based on Rule 1 (Line 2). By Rule 2, the algorithm starts from 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. 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 effectively reduced to that of the traditional fixed-priority system.

V. (α, β)-pair Response Time Analysis

With the proposed schedule and CPC model, this section presents a new response time analysis that explicitly accounts for the parallel workload (i.e., α) and applies α as a safe reduction on the interfering workload that can delay λ^*. In addition, we highlight that although the proposed schedule assigns explicit node priority, the critical path first execution (i.e., CPFE) is a fundamental property to maximise parallelism (see Theorem 1) and is adopted in many existing methods [15], [11], [14]. For generality, the proposed analysis assumes CPFE and allows any scheduling order for non-critical nodes. That is, compared to the traditional analysis [12], [13], this analysis provides an improved bound for all schedules based on CPFE. The analysis does not assume the explicit execution order is known in advance. In Section V-C, we extend the proposed analysis for scheduling methods with an explicit order known a priori (e.g., [17], [11] and the proposed schedule) with minor modifications. Table II summarises the notations introduced in the constructed analysis.

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

In the CPC model, the critical path of a DAG task is transferred to a set of sequential providers Θ^*. A provider $\theta_i^* \in \Theta^*$ can start if and only if the previous provider θ_{i-1}^* and its consumers $F(\theta_{i-1}^*)$ have finished executions (Figure 2(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^* (Figure 2(c)).

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 $f(\theta_i^*)$ [13], [12].

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 $\lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) + \beta_i \rceil$.*

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 $\lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) \rceil + \beta_i$, for a system with m cores. This is proved in [13], [12]. Note, as β_i is accounted for explicitly, it is removed from the interfering workload to avoid repetition. □

Table II: Notations introduced in the proposed (α, β)-pair response time analysis

Notation	Description
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 forms 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.

Based on Lemma 2, the response time analysis for a DAG task can be formulated 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.

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

Compared to the traditional analysis [13], [12], 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 $\lceil \frac{1}{m}(W - L) \rceil > \sum_{\theta_i^* \in \Theta} \lceil \frac{1}{m}(W_i - L_i - \alpha_i - \beta_i) \rceil + \beta_i$, a tighter bound can be obtained. That is, the proposed analysis does not always dominate the traditional bound. Therefore, we take $\min\{R, L + \lceil \frac{1}{m}(W - L) \rceil\}$ as the final analytical bound.

B. 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 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 $\lvert \cdot \rvert$ 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 $v_j \in \lambda^* \vee \lvert\Lambda_{\mathcal{C}(v_j)\setminus\lambda^*}\rvert < m-1$.*

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 $\lvert\Lambda_{\mathcal{C}(v_j)\setminus\lambda^*}\rvert < m-1$. The concurrent nodes that can interfere v_j on $(m-1)$ cores is $\mathcal{C}(v_j)\setminus\lambda^*$. Given that the number of paths in $\mathcal{C}(v_j)\setminus\lambda^*$ 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) [11]. The condition $\lvert\Lambda_{\mathcal{C}(v_j)\setminus\lambda^*}\rvert < m-1$ 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.

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

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 $anc(v_j)$ [11], as given in Equation 4. Note that this equation only applies to non-critical nodes v_j with $\lvert\Lambda_{\mathcal{C}(v_j)\setminus\lambda^*}\rvert \geq m-1$.

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

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.

$$f(\theta_i^*) = \max_{\forall_{v_j \in \theta_i^*}} \left\{ f(v_j) \right\} \quad (5)$$

$$f(F(\theta_i^*)) = \max_{\forall_{v_j \in F(\theta_i^*)}} \left\{ f(v_j) \right\} \quad (6)$$

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.

$$\begin{aligned} \alpha_i = & \sum_{v_j \in V_a} \left\{ C_j \right\} + \sum_{v_j \in V_b} \left\{ f(\theta_i^*) - \left(f(v_j) - C_j \right) \right\}, \\ & \forall v_j \in F(\theta_i^*) \cup G(\theta_i^*), \\ & V_a = \{v_j \lvert f(v_j) \leq f(\theta_i^*)\} \\ & V_b = \{v_j \lvert f(v_j) > f(\theta_i^*) \wedge f(v_j) - C_j < f(\theta_i^*)\} \end{aligned} \quad (7)$$

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 1) it finishes before θ_i^*, i.e., $f(v_j) \leq f(\theta_i^*)$, or 2) 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 $len(\lambda_{v_a} \cup v_c) \geq len(\lambda_{v_b} \cup v_c)$ [11]. 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 $f(\theta_i^*)$.

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

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 [20], 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.

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 $f'(v_k) \leq f(v_k), \forall v_k \in F(\theta_i^*) \cup G(\theta_i^*)$. Accordingly, by Definitions 1 and 2 we have $\alpha_i' \geq \alpha_i$ and $\beta_i' \leq \beta_i$ and hence, leads to 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., sustainable.

C. 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 [11]. 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 node priority, the interfering nodes of v_j on $m-1$ cores can be effectively reduced to 1) nodes in $\mathcal{I}(v_j)$ that have a higher priority than p_j [11], and 2) $m-1$ nodes in $\mathcal{I}(v_j)$ that have a lower priority and the highest WCET due to the non-preemptive schedule [10]. 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 [11] and [10]. For simplicity, we take the $(m-1)$ low priority nodes as a safe upper bound. A finer ILP-based approach is available in [10] to precisely compute this blocking. In addition, if node-level preemption is allowed, $\mathcal{I}^e(v_j)$ is further reduced to $\{v_k | p_k > p_j, v_k \in \mathcal{I}(v_j)\}$.

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

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 [11]. 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}.

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

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 analysis for 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-A). 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 priority, in which $\mathcal{I}^e(v_j) \subseteq \mathcal{I}(v_j)$ and $\mathrm{I}_{v_e} \leq W_i - L_i - \alpha_i - \beta_i$.

In addition, we note that the proposed analysis does not strictly dominate the analysis in [11] for a particular schedule, but can provide a more accurate result in the general case (see results in Section VII-A). In practice, the bound in [11] can be used as a safe upper bound for the proposed analysis, to provide the most accurate known worst-case approximations.

VI. Extension to Support Multiple DAGs

In this section, we extend the proposed scheduling and analysing methods to allow the general sporadic task model with n DAG tasks $\Gamma = \{\tau_1, ..., \tau_n\}$, in which each task τ_x is assigned a unique deadline monotonic priority P_x.

With multiple DAG tasks, the schedule follows the principle of *highest priority task (P_x) first* and then within a task *highest priority node (p_j) first*, for all DAGs tasks and nodes that are ready to release. With a fully non-preemptive DAG-level scheduling, the highest priority task in the ready queue is always scheduled to execute after the currently-executing task is finished. 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. We acknowledge this schedule is not work-conserving, and can lead to certain cores being idle with ready tasks await execution. This would lead to a longer worst-case response time. However, it allows the currently executing task to concentrate the available resources on nodes that form the critical path.

A DAG task τ_x can be delayed by all jobs of high priority tasks released during τ_x's busy period and one job of the low priority task that has the highest completion time. Let $R_x^\diamond$ denote the worst-case response time of τ_x in the multi-DAG case. $R_x^\diamond$ is given by Equation 14, in which R_x gives the worst-case completion time of τ_x in the single-DAG case (by Equation 2), $lp(x)$ returns all tasks with a priority lower than P_x and $hp(x)$ denotes τ_x's higher priority tasks. As a ready task is released after the currently-executing task is completed, the worst-case delay from a job in an interfering task τ_y is effectively bounded by R_y by Equation 2.

$$R_x^\diamond = R_x + \max_{\tau_y \in lp(x)} \left\{ R_y \right\} + \sum_{\tau_y \in hp(x)} \left\lceil \frac{R_x^\diamond}{T_y} \right\rceil R_y \tag{14}$$

Finally, we note that by starting the next task in the ready queue during the "fan-in" phase (in which the parallelism of the DAG decreases monotonically until finished) of the current task, a reduced overall makespan for all tasks can be achieved while not affecting the current task. This is the same principle as used in processors for in order pipeline execution and proven analysis exists for bounding its execution [21]. This will not jeopardise the analysis as a release earlier than expected cannot cause a node to finish later than the worst-case bound. However, this complicates the scheduling and requires an online analysis that identifies the fan-in phase of each DAG, which may not always be feasible in real-world applications. In addition, analysing this early release can further complicate the (α, β)-pair analysis, due to extra offsets between the start of the critical path and non-critical nodes of the task. Notably, with [2], multi-DAGs with different periods can be described as a single periodic DAG, so the proposed analysis can be directly applied. However, this is out of the scope of this paper and is postponed to future research.

VII. Evaluations

The objectives of this evaluation are multifold: (1) to demonstrate the scheduling and analysis (*rta-cpf* in Section V-A and *rta-cpf-eo* in Section V-C) improves the worst-case makespan (using the classic bound as reference); (2) to establish the conditions in which the proposed methods lead

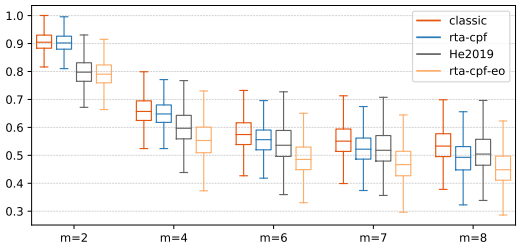


Figure 3: DAG worst-case makespan using analytical methods with varied number of cores (m).

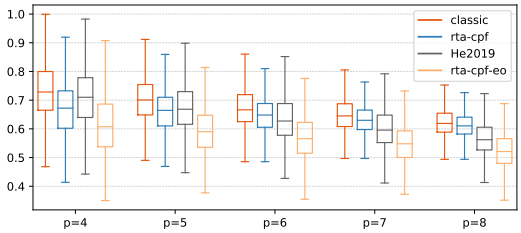


Figure 4: Sensitivity of parallelism parameter when $m = 4$.

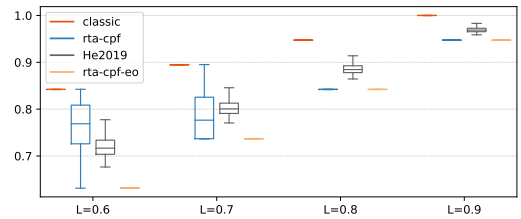


Figure 5: Sensitivity of critical path ratio when $m = 2$, $p = 8$.

to an improved makespan; (3) to demonstrate the proposed execution order (*EO*) improves schedulability and the proposed analysis tightens the worst-case bounds; and (4) to evaluate the improvement through schedulability tests in multi-DAG cases. The proposed node ordering, *EO*, is compared with He et al. 2019 [11] (denoted as He2019 hereafter) in which a node priority assignment is proposed alongside the analysis.

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*

The experiment evaluated the performance scaled with the number of cores (m). For each configuration (task and system setting), 1,000 trials are applied on the compared methods. Each trial generates one DAG task randomly. The normalised worst-case makespan is used as the indicator.

Observation: Figure 3 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, *rta-cpf* becomes effective (starting from $m = 6$) and outperforms the classic bound, e.g., by 15.7% and 16.2% in average (and up to 31.7% and 32.2%), with $m = 7$ 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, *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 VII-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 He 2019, and outperforms He2019 with $m = 8$. This observation further demonstrates the effectiveness of the proposed analysis.

B. *Sensitivity of DAG properties on the evaluated methods*

From the result in Section VII-A, it is not straightforward to understand how DAG properties would impact on the worst-case makespan. To accommodate this, 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): 1) DAG parallelism (the maximum possible width when generating the randomised DAG), p; and 2) DAG critical path ratio to the total workload, $\%L$, where $\%L = L/W \times 100\%$.

Observation: Figure 4 shows the worst-case makespan of the proposed methods with varied values of the parallelism parameter (with $m = 4$). First, 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

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

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 to 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 5 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 case. In addition, we observed that all the tested parameters m, p, $\%L$ have 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 to 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. In a similar fashion to *rta-cpf*, *rta-cpf-eo* demonstrates better performance with the increase of $\%L$. However, due to its explicit execution order, *rta-cpf-eo* demonstrates much stronger performance than *rta-cpf* and is not affected with an impact from parameter p.

C. Effectiveness of the proposed schedule and analysis

In this experiment, the proposed priority assignment is compared against the state-of-the-art node-level priority assignment method, i.e., He2019. In addition, we demonstrate the worst-case makespan with the priority assignment considered. The purpose is to demonstrate the improved worst-case scenario achieved by the priority assignment. 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.

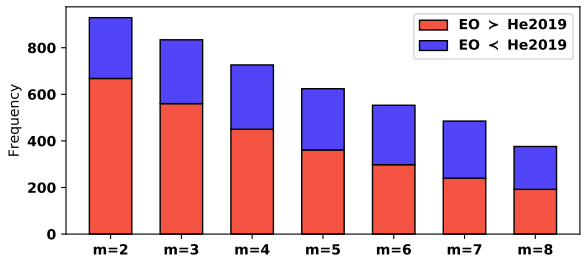


Figure 6: Proposed priority ordering v.s. the ordering in He 2019, grouped by the number of cores (m). $p = 8$. "$\succ$" means outperform and "$\prec$" means the vice versa.

Observation: Figure 6 reports the comparison of the proposed ordering method and the method in He2019, 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-C) 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 orders become less significant to affect the final makespan.

Table III 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 IV 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 IV is a categorical value in (negligible effect, small effect, medium effect and large effect) to reflect the scientific significance [22]. 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, He2019. The *Magnitude* gives further evidence of the benefits of our

Table III: 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

Table IV: Advantage cases numbers and scientific significance in node-level priority assignment – EO and He2019 ordering both 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

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.

Similarly, we have done a comparison of our analysis and the analysis in He2019 by applying the same ordering on both methods and found consistent results (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. Schedulability test with multi-DAGs

To further evaluate priority assignment, we tested the schedulability of random generated multi-DAG tasksets. The experiment is setup as follows: the number of cores is fixed to $m = 6$. The total utilisation of all DAG tasks (averaged on per core) ranges from 0.1 to 1.0, with a step size of 0.05. The utilisation of a DAG task within a taskset is generated through the UUniFast-discard algorithm [23]. The utilisation of each DAG should be less than m, otherwise it is discarded.

A taskset has 10 DAG tasks, and each DAG is generated randomly in the same way as introduced earlier. The periods of DAG tasks are randomly generated for $T_i \in (1000, 2000)$, and deadlines are equal to periods. The execution times of the nodes within a DAG task are then generated based on its workload $W_i = U_i/T_i$. Schedulability for multi-DAGs is tested using Equation 14, in which R_i is calculated using *rta-cpf-eo* analysis and He2019 analysis, respectively. The schedulability of random execution, i.e., without node-level orders is evaluated by the classic response time equation. The priorities are assigned to DAGs based on the deadline-monotonic policy.

Observation: As given in Figure 7, the method *random* gives a reference bound with nodes in each DAG scheduled randomly. From the results, the proposed scheduling and analysing methods provides better system schedulability than that of the state-of-the-art in most cases (i.e., for $\sum U/m = 0.30 - 0.45$). The results are consistent with the single DAG case as DAG tasks are executed in a non work-conserving manner (i.e., one task at a time), in the priority order. Table V reports the detailed schedulability results. From this table, the proposed methods outperform the state-of-the-art up to 53.3% when $\sum U/m = 0.35$.

Table V: Schedulable tasksets (%) to the target utilisation $\sum U/m$ (averaged on per core)

$\sum$ U	0.2	0.25	0.3	0.35	0.4	0.45	0.5
Random	100	99.9	87.4	9.0	0.4	0.3	0.2
EO	100	99.9	99.6	80.0	13.0	0.7	0.2
He2019	100	99.9	95.2	26.7	0.7	0.3	0.2

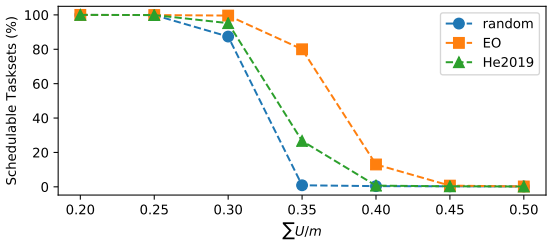


Figure 7: Schedulability v.s. target averaged total utilisation per core for multi-DAGs (when $m = 6$, $p = 8$).

Summary: In these experiments, we have shown that the proposed worst-case makespan analysis and the priority assignment can generally improve the schedulability by tightening the worst-case bound. The effectiveness of the method is also shown by the improved number of schedulable tasks with multi-DAGs compared with the existing approaches.

VIII. Concluding Remarks

In this paper, a rule-based scheduling method is proposed which maximises node parallelism to improve the schedulability of single DAG tasks. Based on the rules, response time analysis is developed that provides tighter bounds than existing analysis for 1) any scheduling method that prioritises the critical path, and 2) scheduling methods with explicit execution order known a priori. We demonstrate that the proposed scheduling and analysing methods outperform existing techniques. In future work, we will focus on further optimisations of the proposed method and extensions to fully support work-conserving schedules for multiple recurrent DAGs.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] G. Buttazzo and A. Cervin, "Comparative assessment and evaluation of jitter control methods," in *Conference on Real-Time and Network Systems*, 2007, pp. 163–172.

[10] 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.

[11] 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.

[12] 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.

[13] R. L. Graham, "Bounds on multiprocessing timing anomalies," *Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *International Workshop on Embedded Software*. Springer, 2002, pp. 334–348.

[22] 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.

[23] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010, pp. 6–11.