applied sciences

Article

DAG Hierarchical Schedulability Analysis for Avionics Hypervisor in Multicore Processors

Huan Yang [1,*], Shuai Zhao [2], Xiangnan Shi [1], Shuang Zhang [3] and Yangming Guo [1,*]

[1] School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China
[2] Department of Computer Science, University of York, York YO10 5GH, UK
[3] Xi'an Aeronautical Computing Technique Research Institute, Xi'an 710068, China
* Correspondence: yhneu2009@163.com (H.Y.); yangming_g@nwpu.edu.cn (Y.G.)

Abstract: Parallel hierarchical scheduling of multicore processors in avionics hypervisor is being studied. Parallel hierarchical scheduling utilizes modular reasoning about the temporal behavior of the upper Virtual Machine (VM) by partitioning CPU time. Directed Acyclic Graphs (DAGs) are used for modeling functional dependencies. However, the existing DAG scheduling algorithm wastes resources and is inaccurate. Decreasing the completion time (CT) of DAG and offering a tight and secure boundary makes use of joint-level parallelism and inter-joint dependency, which are two key factors of DAG topology. Firstly, Concurrent Parent and Child Model (CPCM) is researched, which accurately captures the above two factors and can be applied recursively when parsing DAG. Based on CPCM, the paper puts forward a hierarchical scheduling algorithm, which focuses on decreasing the maximum CT of joints. Secondly, the new Response Time Analysis (RTA) algorithm is proposed, which offers a general limit for other execution sequences of Noncritical joints (NC-joints) and a specific limit for a fixed execution sequence. Finally, research results show that the parallel hierarchical scheduling algorithm has higher performance than other algorithms.

Keywords: multicore; avionics hypervisor; DAG; hierarchical; parallel scheduling

Citation: Yang, H.; Zhao, S.; Shi, X.; Zhang, S.; Guo, Y. DAG Hierarchical Schedulability Analysis for Avionics Hypervisor in Multicore Processors. *Appl. Sci.* **2023**, *13*, 2779. https://doi.org/10.3390/app13052779

Academic Editor: Teh-Lu Liao

Received: 6 January 2023
Revised: 15 February 2023
Accepted: 20 February 2023
Published: 21 February 2023

1. Introduction

Multicore processors in avionics systems are being used to meet the increasing demands for performance and energy efficiency [1–3]. In the design of a real-time multi-core system, to avoid the interference of system resources, physical isolation technology is often used to isolate system resources. Virtualization technology can deploy subsystems with different functions on virtual machines running on the same hardware platform, which provides a more flexible method for system isolation and resource management. For virtual machines, the hypervisor is responsible for managing virtual machines and shielding the implementation details of the underlying hardware. Flexible strategies are adopted to effectively allocate the underlying hardware resources to virtual machines, to meet the resource requirements of every virtual machine. For real-time multi-core systems, virtualization technology can well meet some urgent requirements in the design of real-time multi-core systems. However, virtualization technology is still a new application direction in the field of real-time embedded systems, and the related research work is still very limited. The main problem is that when virtualization technology is introduced into real-time embedded systems, the response performance of real-time operating systems running on virtual machines is easily affected by the virtualization software layer. The semantic gap caused by the introduction of the virtualization layer makes it difficult for virtual machine monitors to perceive the application types of upper-level virtual machines. It hinders the virtual machine monitor from effectively allocating hardware resources according to the requirements of the upper application, thus it cannot provide a good guarantee for applications with high real-time requirements. Important research to ensure the quality of service is to dynamically allocate enough resources to the real-time VM and make it real-time.

When deploying a virtualization environment in avionics systems, the virtualization system needs to adopt a hierarchical scheduling framework. (i) Scheduling of VCPUs on PCPUs; (ii) Scheduling of real-time tasks in VCPUs (Figure 1) [4–6]. The scheduling levels at these two layers must meet the real-time guarantee, thus fulfilling the real-time guarantee of the whole system. However, classical real-time scheduling is no longer applicable to this hierarchical scheduling structure. A hierarchical scheduling framework needs to provide hierarchical resource sharing and allocation strategies for different scheduling services under different scheduling algorithms. The hierarchical scheduling framework can be expressed as a tree with a joint (hierarchical) structure. Every joint can be expressed as a scheduling model and the resources are to be allocated from the parent joint to the child's joint. The resource allocation from the parent joints to the child's joints can be regarded as a scheduling interface from the parent joint to the child. Therefore, the hierarchical real-time scheduling problem can be transformed into the schedulability analysis of the scheduling interfaces of the child and parent joints [7–9].

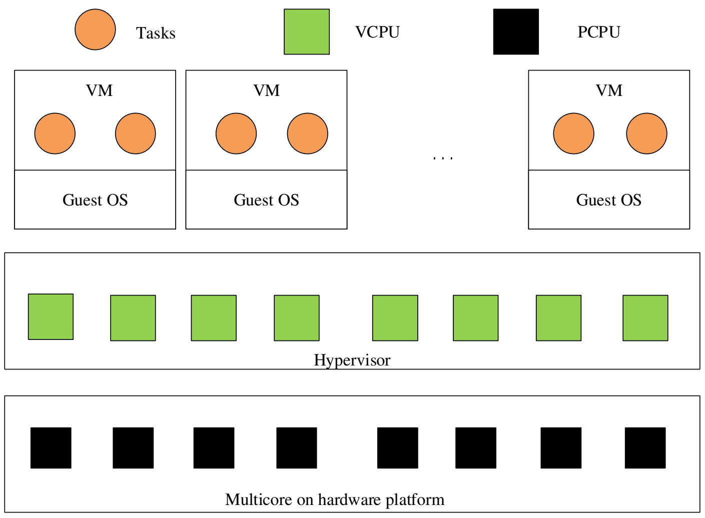

Figure 1. Virtualization hierarchical scheduling framework.

To solve this problem, in recent years, some researchers have begun to pay attention to the parallel hierarchical scheduling of multicore processors. At present, most of the research on program parallelization in multi-verification focuses on the DAG task model [10]. DAG task model can describe the execution dependencies of task threads, such as parallel execution and serial execution. The sequential synchronous parallel task model is a stricter model for task behavior in the parallel DAG model. In this model, tasks can be divided into segments executed in series, every part can contain any number of parallel threads, and whole threads have to synchronize at the end of the segment. The existing DAG real-time task scheduling algorithms have two types: global scheduling and federated scheduling [11,12]. In global scheduling, multiple DAG tasks enjoy together whole multicores, so the overall resource utilization of the system may be high [5–10]. Due to complex interference between tasks, global scheduling has great inaccuracy. In federated scheduling, every task is assigned to some dedicated processors, where it can be executed without interference from other tasks. The scheduling analysis is relatively simple, but it wastes resources [12].

The main contribution: a single-cycle non-preemptive DAG task runs on an isomorphic multicore processor. By making full use of joint-level parallelism and inter-joint dependency, which are the essence of topology, they decrease the maximum CT and offer a tight and safe boundary for the maximum CT. This paper presents a new algorithm of the parallel hierarchical scheduling method that has two sections: (i) DAG parallel schedulability analysis, and (ii) hierarchical scheduling in the hypervisor.

The rest of this article is organized as follows. In Section 2, the system and task model are introduced. Section 3 describes the state-of-the-art approaches in DAG scheduling and analysis with a motivational example. The proposed scheduling method, CPCM, and the new response time analysis are explained in Section 4. The hierarchical scheduling in the hypervisor is provided in Section 5. Results are given in Section 6. Finally, Section 7 concludes this article.

2. Preliminaries

2.1. System Model

The system defines parallel task sets which are scheduled on a multi-core processor, the task set is defined as τ consisting of tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and the hardware platform includes M homogeneous processors. Each task is represented by a DAG with p joints and edges connecting these joints.

2.2. Task Model

Any DAG task is τ_i= $\{D_i, T_i, G_i\}$, where D_i is the constrained relative deadline, T_i is the minimum inter-arrival time, within $D_i \leq T_i$, and G_i is the graph defining a series of activities forming the task. The graph is characterized by $G_i = (V_i, E_i)$ where V_i is a series of joints and $E_i \subseteq (\overrightarrow{V_i V_j})$ is a series of directed edges connecting any two joints. Every joint $v_i \in V_i$ is the computation unit that must be executed unceasingly and is defined by WCET (Worst Case Execution Time), namely C_i.

Either v_j, v_k form a directional edge, only if v_j has finished and v_k begins to be executed, v_j is a predecessor of v_k, and v_k is a successor of v_j. Each joint v_j has a $predecessor(v_j)$ and a $successor(v_j)$, $predecessor(v_j) = \{v_k \in V \big| (v_j, v_k) \in E\}$ and $successor(v_j) = \{v_k \in V \big| (v_j, v_k) \in E\}$. Joints that are either directly or transitively predecessors and successors of a joint v_j are termed as its ancestors $ancestors(v_j)$ and descendants $descendants(v_j)$ respectively. The joint v_j with $predecessor(v_j) = \varnothing$ or $successor(v_j) = \varnothing$ is referred to as the sink v_{sink} or source v_{source}. In order not to lose generality, suppose DAG has one sink and source joint. Joints that can execute concurrently with v_j are given by $C(v_j) = \{v_k \big| v_k \notin (ancestors(v_j) \cup descendants(v_j)), \forall v_k \in V\}$ [13].

DAG tasks have basic characteristics. Firstly, an arbitrary path $\delta_a = \{v_s, \cdots, v_e\}$ is a joint sequence in V and follows $(v_k, v_{k+1}) \in E, \forall v_k \in \delta_a / v_e$. The path 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_{source} and the sink v_{sink}, $length(\delta_a) = \sum_{\forall v_k \in \delta_a} C_k$ provides the length of δ_a. Secondly, the longest complete path is referred to as the critical path (CP) δ^*, and its length is denoted by L, where $L = \max\{length(\delta_a), \forall \delta_a \in \Lambda_V\}$. Joints in δ^* are referred to as the critical joints. Other joints are referred to as NC-joints (NC-joints), denoted as $V^\neg = V / \delta^*$. Finally, the workload W is the sum of a task's WCETs, $C = \sum_{\forall v_k \in V} C_k$. The workload of all NC-joints is called the non-critical workload.

Figure 2a depicts a DAG task with $V = \{v_1, v_2, \cdots, v_9\}$. The number in the upper left corner of every joint provides WCET, for example $C_6 = 4$. For v_6 have $predecessor$ $(v_6) = \{v_2, v_3\}$, $ancestors(v_6) = \{v_1, v_2, v_3\}$, $successor(v_6) = descendants(v_6) = \{v_9\}$ and $C(v_j) = \{v_2, v_3\}$. L = 14, C = 30, $\delta^* = \{v_1, v_2, v_6, v_9\}$, $v_{source} = v_1$, $v_{sink} = v_9$.

2.3. Work-Conserving Schedulability Analysis

When there is a pending workload, the scheduling algorithm never idles the processor, and it is called a work-conserving algorithm [14]. A general bound is provided, which catches the worst-case (WC) response time of globally scheduling tasks using any work-saving algorithm [15]. This analysis is then formalized in the DAG task in Formula (1) [16].

R_i is the response time of τ_i, m is the number of processors, $I_{i,j}$ is the interference from a high priority DAG task τ_j, and $hp(i)$ is whole high priority tasks of τ_i.

$$R_i = [\textstyle\sum_{\tau_j \in hp(i)} I_{i,j}] + \lceil (C_i - L_i)/m \rceil + L_i \tag{1}$$

Figure 2b shows some execution scene of DAG in a quad-core processor. For casually scheduled joints, there may be a series of 420 execution scene, the maximum CT is less than 14, and gives $R = (C - L)/m + L = (30 - 14)/4 + 14 = 18$. They take less than 18. Based on the above, the paper puts forward a new algorithm to decrease the makespan of the runtime.

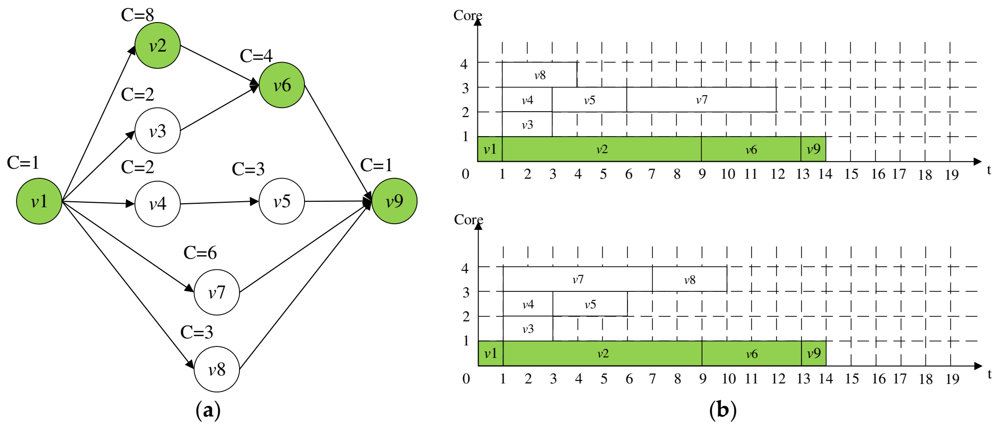

Figure 2. Example of a DAG task; (**a**) A DAG task (**b**) Execution scene.

3. Related Work

For multicore processors with global solutions, the existing scheduling methods aim to decrease the maximum CT and tighten analysis limit of the worst case. They can be divided into joint-based [17] or slice-based [18,19] methods. Chip-based scheduling implements joint-level preemption and divides every joint into many small computing units. The slice-based method can promote the joint-level parallelism, but to achieve an improvement, it is necessary to control the numeral of preemption and migration. The joint-based method provides a better general solution by generating a clear joint execution sequence based on heuristics derived from the spatial or temporal characteristics of DAG [20].

Two of the latest joint-based approaches are described below. An exception-free non-preemptive scheduling method for single-cycle DAG is proposed, which always executes the longest ready joint of WCET to improve the parallelism. This prevents an exception when a joint executes less than its WCETs, which will cause the execution order to be different from the plan. This is achieved by ensuring that the joints are executed in the same order as the offline simulation. However, if the dependency between joints is not considered, this scheduling cannot minimize the delay of DAG completion. This algorithm leads to a scene with a maximum CT of 14, in which the NC-joint v_3 prolongs DAG completion due to a delayed start in Figure 2.

A new Response Time Analysis (RTA) method is proposed, which is superior to the traditional method when the execution sequence of joints is known [16–18]. That is, joint v_j can only lead to delays from concurrent joints scheduled before v_j. A scheduling algorithm is given, in which (i) the CP is always executed first, and (ii) the first is always the intermediate interfering joint. The novelty lies in considering the topology and path length in DAG, and providing the analysis compared with our method [19,20]. However, the parallel joints are scheduled according to the length of the longest full path, and the joints in the longest full path are scheduled first. Heuristic algorithms do not rely on perception, which will decrease parallelism and thus prolong the final CP.

4. Schedulability Analysis

4.1. Scheduling

Formula (1) shows that minimizing the latency from NC-joints to CPs effectively decreases the maximum CT of DAG. To support this point, a CPCM is recommended to take full advantage of joint dependency and parallelism. A scheduling algorithm to maximize the parallelism of joints is proposed. This is achieved through rule-based priority allocation, in which rules are designed to statically assign priority to every joint: (i) Always give priority to the CP priority, (ii) Rule 2 and 3 to maximize parallelism, (iii) minimize the latency of the CP. The algorithm is universally applicable to DAG of any topology. It adopts homogeneous processors, but there is no limit to the number of processors [21–23].

4.1.1. Concurrent Parent and Child Model

The CPCM includes two key parts. (i) The CP is divided into a set of consecutive sub-paths based on the potential delay that can happen (in Figure 3a). (ii) With every sub-path, CPCM distinguishes NC-joints, which can run in parallel with the sub-path and prolong the start of the next sub-path on the basis of priority constraints (in Figure 3b,c).

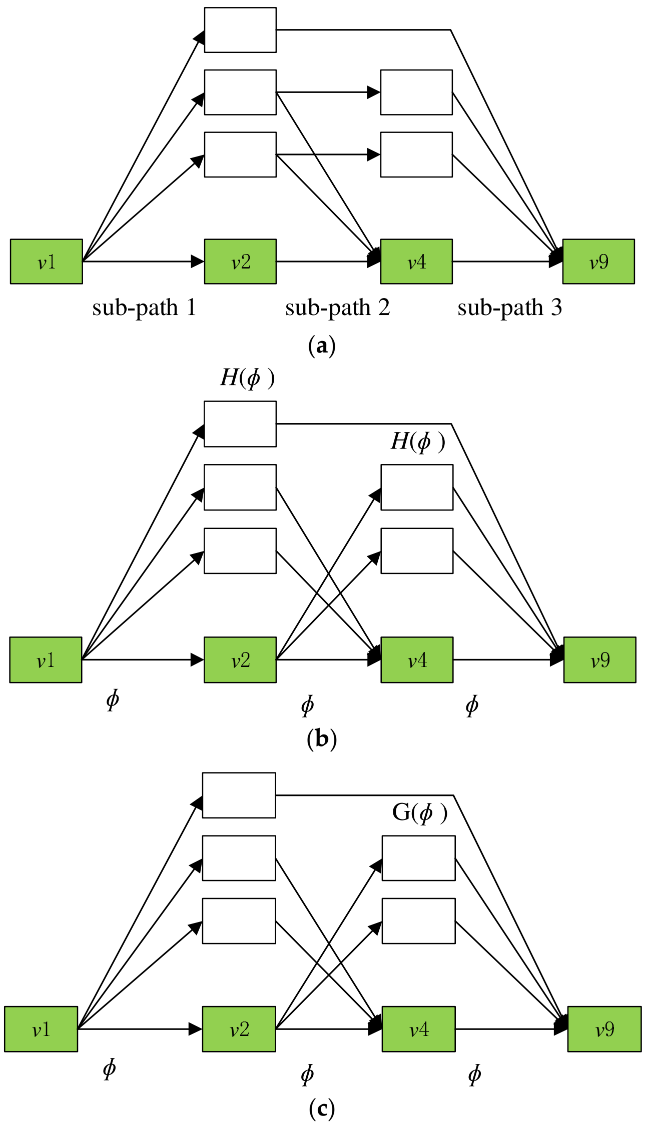

Figure 3. The CPCM of a DAG. (**a**) the CP is divided into a set of consecutive sub-paths based on the potential delay it can incur. (**b**) execute in parallel with the sub-path. (**c**) delay the start of the next sub-path, based on precedence constraints.

The intuition of the CPCM is: when the critical path is executing, it utilizes 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 joints in parallel is termed as the capacity, which is the length of the critical path. Note that non-critical joints that utilize this capacity to execute cannot cause any delay to the critical path. The sub-paths in the critical path are termed capacity parents Φ^* and all non-critical joints are capacity children Φ. For each parent $\phi_i^* \in \Phi^*$, it has a set of children $H(\phi_i^*)$ that can execute using ϕ_i^*'s capacity as well as delay the next parent ϕ_{i+1}^* in the critical path.

Algorithm 1 shows the process for constructing the CPCM. Starting from the scratch in δ^*, capacity parents are formed by analyzing joint dependency between the CP and NC-joints. A parent ϕ_i^*, its joints should run consecutively without prolong from NC-joints according to dependency. Every joint in ϕ_i^*, other than the scratch, only owns a predecessor which is the previous joint in ϕ_i^*. Three capacity parents are identified in Figure 3b.

Algorithm 1: Concurrent Parent and Child Model algorithm $CPCM(G, \delta^*)$

```
**Input:** $G = (V, E)\ \delta^*$
**Output:** $\Phi^*, H(\phi_i^*), G(\phi_i^*), \forall\phi_j^* \in \Phi^*$
**Specifications:** $\delta^*, V^\neg = V/\delta^*$
**if** $\Phi^* = \varnothing$
/* distinguishing capacity parents */
**for** every $v_j \in \delta^*$
    $\phi_i^* = \{v_j\}; \delta^* = \delta^*/v_j$
    **while** $predecessor(v_{j+1}) = \{v_j\}$ **do**
        $\phi_i^* = \phi_i^* \cup \{v_{j+1}\}; \delta^* = \delta/v_j$
    **end**
    $\Phi^* = \Phi^* \cup \phi_i^*$
**end**
/* distinguishing capacity children */
for every $\phi_i^* \in \Phi^*$
    $H(\phi_i^*) = ancestors(\phi_{i+1}^*) \cap V^\neg$
    $G(\phi_i^*) = \cup_{v_j \in H(\phi_i^*)}\{C(v_j) \cap V^\neg\}$
    $V^\neg = V^\neg/H(\phi_i^*)$
**end**
**return** $\Phi^*, H(\phi_i^*), G(\phi_i^*), \forall\phi_i^* \in \Phi^*$
```

Every $\phi_i^* \in \Phi^*$, its children $H(\phi_i^*)$ are determined to the joints that can run synchronously to ϕ_i^*, and prolong the start of ϕ_{i+1}^*. Joints in $H(\phi_i^*)$ that finish after ϕ_i^* will prolong the start of ϕ_{i+1}^*. Joints in $H(\phi_1^*)$ can prolong ϕ_2^* if they are finished later than ϕ_1^* in Figure 3b. The CPCM shows details of the possible latency attributed to NC-joints on the CP.

In addition, a child $v_j \in H(\phi_i^*)$ can start before, synchronized with, or after the start of ϕ_i^*. With the latter two, they will just use the capacity of ϕ_i^*. Children of earlier releases can run at the same time as some previous parents, thus interfering with their children and causing indirect delays to parents. With a parent ϕ_i^*, $G(\phi_i^*)$ is the joints that fall within the child groups of later parents, but which can run with ϕ_i^* in parallel. Joints in $G(\phi_1^*)$, $G(\phi_2^*)$ fall within $H(\phi_1^*)$, $H(\phi_2^*)$ in Figure 3c. However, on the basis of the precedence constraints, it can be executed in parallel with ϕ_1^* and ϕ_2^* respectively.

Using the CPCM, the DAG is converted to a series of capacity parents and children, $O(|V|+|E|)$ is time complexity. The CPCM supplies complete information about possible delays of NC-joints on the CP. Every parent ϕ_i^*, joints in $H(\phi_i^*)$ can use a capacity of $length(\phi_i^*)$ on every of $(m-1)$ processors to run in parallel as lead, storing possible latency.

Reanalyze in Figure 2a, the CP has three parents $\phi_1^* = \{v_1, v_2\}$, $\phi_2^* = \{v_6\}$ and $\phi_3^* = \{v_9\}$, as the delay from the NC-joint just appears at the header of the parents. For every parent, $H(\phi_1^*) = \{v_3\}$, $H(\phi_2^*) = \{v_5, v_7, v_8\}$ and $H(\phi_3^*) = \varnothing$. In short, whole joints in $H(\phi_2^*) = \{v_5, v_7, v_8\}$ can start before ϕ_2^* prolonging the execution of $H(\phi_1^*)$ and the start of ϕ_2^*. $G(\phi_1^*) = \{v_5, v_7, v_8\}$, $G(\phi_2^*) = G(\phi_3^*) = \varnothing$.

Parallel and disruptive workloads of the capacity parent are now formalized. $h(\cdot)$ is the completion time of a parent ϕ_i^* or a child joint v_j, $L_i = length(\phi_i^*)$, ϕ_i^* is the length and the total workload of ϕ_i^* is $C_i = \sum_{v_k \in H(\phi_i^*)} \{C_k\} + \sum_{v_k \in G(\phi_i^*)} \{C_k\} + L_i$, $H(\phi_i^*)$ and $G(\phi_i^*)$. The terms parallel and disruptive workload of a parent ϕ_i^* is formally defined. $\sum_{\phi_i^* \in \Phi} C_i \geq C$ as a child can be accounted for more than once if it can run concurrently with multiple parents.

Definition 1. *The parallel workload α_i of ϕ_i^* is the workload in $C_i - L_i$ that can run starting at the time instant $h(\phi_i^*)$.*

Definition 2. *The interfering workload of ϕ_i^* is the workload in $C_i - L_i$ that runs after the time instant $h(\phi_i^*)$. With a parent ϕ_i^*, its interfering workload is $C_i - L_i - \alpha_i$.*

Theorem 1. *For parents ϕ_i^* and ϕ_{i+1}^*, the workload in C_i that can prolong the start of ϕ_{i+1}^* is $C_i - L_i - \alpha_i$ at best.*

Proof. Based on the CPCM, the start of ϕ_{i+1}^* depends on the finish of both ϕ_i^* and $H(\phi_i^*)$, which is $\max\{f(\phi_i^*), \max_{v_j \in H(\phi_i^*)} h(v_j)\}$. By Definition 1, α_i will not cause any delay as it always finishes before $h(\phi_i^*)$, and hence, the Theorem follows. Note that, although $G(\phi_i^*)$ cannot delay directly, it can delay on nodes in $H(\phi_i^*)$, and in turn, causes an indirect delay to ϕ_{i+1}^*. $\square$

4.1.2. The CP Priority Execution (CPPE)

The CP is regarded as a series of capability parents. It can be safely said that every full path could be regarded as the parents, which provides the time interval of its path length for other joints to run in parallel. However, since the CP offers the maximum capacity, the maximum total parallel workload can be achieved. It offers the basis for minimizing the disturbing workload on the whole CP.

Theorem 2. *For a schedule S with CPPE and a schedule S' that prioritizes a random complete path over the CP, the total parallel workload of parents in S is always equivalent to or higher than that of S', $\alpha \geq \alpha'$.*

Proof. First, suppose the length of parent δ_i^* is shortened by Δ after the change from S to S'. The same reduction applies on its finish time, i.e., $h'(\delta_i^*) = h(\delta_i^*) - \Delta$. Because nodes in δ_i^* are shortened, the finish time $h(v_j)$ of a child node $v_j \in H(\delta_i^*) \cup G(\delta_i^*)$ 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 δ_i^* belong to $C(v_j)$) to Δ (if all such nodes belong to $predecessor(v_j)$). By Definition 1, a child $v_j \in H(\delta_i^*) \cup G(\delta_i^*)$ can contribute to the α_i if $h(v_j) \leq h(\delta_i^*)$ or $h(v_j) - C_j \leq h(\delta_i^*)$. Therefore, α_i cannot increase in S', as the reduction on $h(\delta_i^*)$ (i.e., Δ) is always equal or higher than that of $h(v_j)$ (i.e., Δ/m or Δ).

Second, let L and L' denote the length of the parent path under S and S' (with $L \geq L'$), respectively. The time for non-critical nodes to execute in parallel with the parent path is L' on each of $m-1$ cores under S'. Thus, a child 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 child can execute in parallel with the parent.

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

Rule 1. $\forall v_j \in \Phi^*, \forall v_k \in \Phi \rightarrow q_j < q_k$.

4.1.3. Exploiting Parallelism and Joint Dependency

Using CPPE, the next goal is to maximize the parallelism of NC-joints and decrease latency of CP completion. On the basis of the CPCM, every parent ϕ_i^* correlates with $H(\phi_i^*)$, $G(\phi_i^*)$. With $v_j \in G(\phi_i^*)$, it could run before $H(\phi_i^*)$ and use the capacity of ϕ_i^* to run, if a high priority is assigned. In this instance, v_j prolongs the finish of $H(\phi_i^*)$ and the start of ϕ_{i+1}^*, squanders the capacity of its parent. Similar results are also acquired, which avoid this latency by first probing the front interfering joints.

Rule 2. $\forall \phi_i^*, \phi_l^* \in \Phi' : i < l \rightarrow \min\limits_{v_j \in H(\phi_i^*)} q_j > \max\limits_{v_k \in H(\phi_l^*)} q_k$.

Consequently, a second allocation Rule is derived to rule the priority of every parent in the child groups. With any two neighboring parents ϕ_i^* and ϕ_{i+1}^*, the priority of any child in $H(\phi_i^*)$ is higher than the children in $H(\phi_{i+1}^*)$. The latency from $G(\phi_i^*)$ on $H(\phi_i^*)$ can be minimized, because whole joints in $G(\phi_i^*)$ fall within children of following parents and are always assigned with a lower priority than joints in $H(\phi_i^*)$ in Rule 2.

Scheduling the child joints in every $H(\phi_i^*)$, concurrent joints with the same lead time are sorted by the length of their longest full path. Nevertheless, on the basis of CPCM, a full path can be divided into some partial paths, and all of these partial paths fall within a subgroup of different parents. With partial paths in $H(\phi_i^*)$, the order of their lengths can be completely reversed from to the order of their full paths. Therefore, this method can prolong the end time of $H(\phi_i^*)$.

In the constructed scheduling, it is guaranteed to assign higher priority to the longer local paths in a dependency-aware way. It will result in the final allocation rule. Symbol $l_j(H(\phi_i^*))$ is the length of the longest local path in $H(\phi_i^*)$ that contains v_j. This length can be calculated by traversing $ancestors(v_j) \cup descendants(v_j)$ in $H(\phi_i^*)$, such as in Figure 2, $l_7(H(\phi_7^*)) = 6$, $l_8(H(\phi_8^*)) = 3$, so v_7 is assigned a higher priority than v_8. Rules 1–3 are applied to the sample, and the better-case schedule with a maximum full-time of 14 is finally obtained.

Rule 3. $v_j, v_k \in H(\phi_i^*) : l_j(H(\phi_i^*)) > l_k(H(\phi_k^*)) \rightarrow q_j > q_k$.

Nevertheless, applying Rule 3 to every $H(\phi_i^*)$ is not enough. Given the DAG structure, every $H(\phi_i^*)$ can form a smaller DAG $G\prime$, and an inner nested CPCM with the longest path in $H(\phi_i^*)$ is the parent. In addition, this process can be applied recursively to build an internal CPCM for every subgroup in the nested CPCM until whole local paths in the subgroup are completely absolute. With every internally CPCM, Rules 1 and 2 should be applied to maximize the capacity and minimize the latency of every subgroup, while Rule 3 is applied only to the independent paths in subgroups to maximize parallelism. This makes complete dependencies between joints and ensures that the longest path in every nested CPCM takes precedence.

Algorithm 2 shows the method of priority allocation based on Rule. The approach starts from the outer-most CPCM $CPCM(G, \delta^*)$, assign the highest priority to whole parent joints according to Rule 1. In accordance with Rule 2, it starts from before $H(\phi_i^*)$ and searches for the longest local path δ_{v_e} in $H(\phi_i^*)$. If there exists a dependency between nodes in δ_{v_e} and $H(\phi_i^*)/\delta_{v_e}$, $H(\phi_i^*)$ is further constructed as an inner CPCM with the assignment algorithm applied recursively. This resolves the detected dependency by dividing δ_{v_e} into a set of providers. Otherwise, δ_{v_e} is an independent local path so that priority is assigned to its nodes based on Rule 3. The algorithm then continues with $H(\phi_i^*)/\delta_{v_e}$. The process goes on until whole joints in V are assigned priority.

Using CPCM, the whole process contains three stages: (i) passing the DAG to CPCM, (ii) statically assigning priority to every point through rule-based priority assignments, and (iii) running DAG through a fixed priority scheduler. By using a priori input DAG, stages (i) and (ii) could be executed off-line, so as to the scheduling cost of runtime can be effectively decreased to it of the traditional fixed priority systems.

Algorithm 2: Priority assignment algorithm $EA(\Phi^*, \Phi)$

```
Input: $\Phi, \Phi^*$
Output: $H(\phi_i^*) = H(\phi_i^*)/\delta_{v_e}$
Parameter: $q, q^{\max}$
Intialise:   $q = q^{\max}, \forall v_j \in \Phi^* \cup \Phi, p_j = -1$
/* Rule 1. */
$\forall v_j \in \Phi^*, q_j = q, q = q - 1$
/* Rule 2. */
for every $\phi_i^* \in \Phi^*$, do
    while $H(\phi_i^*) \neq \varnothing$ do
     /* Seek for the longest partial path in $H(\phi_i^*)$. */
    $v_e, v_j \in H(\phi_i^*)$:
        $v_e = \text{argmax}\{l_e(H(\phi_i^*)) \big| successor(v_e) = \varnothing\}$
        $\delta_{v_e} = v_e \cup \delta_{v_j}, \text{argmax}\{l_j(H(\phi_i^*)) \Big| \forall v_j \in predecessor(v_e)\}$
     if $\Big|predecessor(v_j)\Big| > 1, \exists v_j \in \delta_{v_e}$ then
        $\{\Phi^{*'}, \Phi'\} = CPCM(H(\phi_i^*), \delta_{v_e})$
        $EA(\Phi^{*'}, \Phi')$
        break
     else
        /* Rule 3. */
       $\forall v_j \in \delta_{v_e}, q_j = q, q = q - 1$
       $H(\phi_i^*) = H(\phi_i^*)/\delta_{v_e}$
      end
    end
end
```

4.2. Analysis of Response Time

According to the above, we will put forward another new Analysis of Response Time, which clearly illustrates the parallel workload α and uses α is a safely decrease for potentially delayed disruptive workloads. In short, it emphasizes that, although the proposed scheduling assigns a clear joint priority, CPPE is a basic attribute to maximize parallelism and has been adopted in many existing algorithms. In general, the CPPE permits any scheduling order of NC-joints. This RTA offers an improved boundary for whole CPPE-based scheduling compared to traditional analysis. The analysis does not assume that a clear run sequence is known in advance. In Section 4.2.3, the proposed analysis of scheduling algorithms was extended with minor modifications by using an explicit order known a priori.

4.2.1. The (α, β)-pair Analysis Formulation

In CPCM, the CP of the DAG task is converted to a series of uninterrupted parents Φ^*. A parent $\phi_i^* \in \Phi^*$ can start, only if the previous parent ϕ_{i-1}^* and its children $H(\phi_{i-1}^*)$ have finished the run (in Figure 3b). In any case, $H(\phi_{i-1}^*)$ can lead to a latency from $G(\phi_{i-1}^*)$ (early-released children that can run concurrently with $H(\phi_{i-1}^*)$), which in turn, delays the start of ϕ_{i-1}^* (in Figure 3c). Based on Definitions 1 and 2, the parallel workload α_i of ϕ_i^* finishes no later than $h(\phi_i^*)$ on $m-1$ processors. After ϕ_i^* completes, the interfering workload then executes on whole processors, in which the latest-finished joint in $H(\phi_i^*)$ gives the earliest starting time to the next parent (if exist). Therefore, bounding this delay requires:

(1) Bounds of parallel workload (α_i);
(2) Bounds of the longest run sequence in $H(\phi_i^*)$ that runs later than $h(\phi_i^*)$, expressed as β_i.

With a random execution order, the WC completion time of β_i imposesd an effective upper bound for the WC completion of workload in $H(\phi_i^*)$ that runs later than $h(\phi_i^*)$. With α_i and β_i expressed, Theorem 2 offers the bounds on the latency ϕ_i^* due to the child joints in $H(\phi_{i-1}^*)$.

Theorem 3. *Two uninterrupted offers ϕ_i^*, ϕ_{i-1}^*, the children joints in $H(\phi_{i-1}^*)$, can prolong ϕ_i^* no more than $\lceil (C_i - \alpha_i - L_i - \beta_i)/m + \beta_i \rceil$.*

Proof. By Definition 2, the interfering workload in $H(\delta_i^*) \cup G(\delta_i^*)$ that can (directly or transitively) delay δ_i^* is at most $C_i - L_i - \alpha_i$. Given the longest execution sequence in $H(\delta_i^*)$ in the interfering workload (i.e., β_i), the worst-case finish time of $H(\delta_i^*)$ (and also β_i) is bounded as $\lceil (C_i - \alpha_i - L_i - \beta_i)/m + \beta_i \rceil$, for a system with m cores. Note, as β_i is accounted for explicitly, it is removed from the interfering workload to avoid repetition. □

On the basis of Theorem 3, the RTA of DAG task can be expressed by Formula (2). As $C_i - L_i - \alpha_i$ starts strictly after $h(\phi_i^*)$, the completion time of both ϕ_i^* and $H(\phi_i^*)$ is bounded by the length of ϕ_i^* and the WCET of β_i. In short, ϕ_{i+1}^* can only start after the finish of ϕ_i^* and whole joints in $H(\phi_i^*)$. Therefore, the final response time of DAG is limited by the sum of the CTs of every parent and its children.

$$R = \sum\nolimits_{\phi_i^* \in \Phi^*} \{ \lceil (C_i - \alpha_i - L_i - \beta_i)/m \rceil + \beta_i + L_i \} \tag{2}$$

Compared with traditional analysis, it can promote the WC response time approximation by tightening the interference on the CP, and the correctness is not damaged. $\sum_{\phi_i^* \in \Phi} \lceil (C_i - L_i - \alpha_i - \beta_i)/m \rceil + \beta_i < \lceil (C - L)/m \rceil$, tighter boundaries could be acquired. Namely, analysis is not always carried out beyond the traditional limits. Hence, $\min\{L + \lceil (C - L)/m \rceil, R\}$ is the final analytical boundary.

4.2.2. Bounding α_i and β_i

For v_j, it can subject to interference (I_j) from the concurrent joints upon arrival. Until constraint $h(v_j)$, first distinguish two particular senses in which the interference of a joint v_j is 0 in Theorem 4, $C(v_j)$ provides v_j's concurrent joints, $\wedge V$ is paths in a given joint set V and $|\cdot|$ returns the size of a given set.

Theorem 4. *According to a schedule with CPPE, joint v_j does not lead to any interference from its concurrent joints $C(v_j)$, if $v_j \in \delta^* \vee \left| \Lambda_{C(v_j)/\delta^*} \right| < m - 1$.*

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

Second, a node $v_j \in V^\neg$ does not incur any interference if $\left| \Lambda_{C(v_j) \setminus \delta^*} \right| < m - 1$. The concurrent nodes that can interfere v_j on $(m-1)$ cores are $C(v_j) \setminus \delta^*$. Given that the number of paths in $C(v_j) \setminus \delta^*$ 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.

Theorem 5. *With respect to the end joint v_e in the longest path of $H(\phi_i^*)$, $h(v_e) = h(H(\phi_i^*))$.*

Proof. Given two paths δ_a and δ_b with length $L_a > L_b$ and a total workload of C, it follows that $h(\delta_a) = L_a + (C - L_a)/m \geq h(\delta_b) = L_b + (C - L_b)/m$, as $h(\delta_a) - h(\delta_b) = L_a - L_b + (L_b - L_a)/m \geq 0$. Therefore, node v_e with $h(v_e) = h(H(\phi_i^*))$ gives the end node of the longest path in the interfering workload. □

Theorem 6. *The leading joint of the end joint v_e in the longest path of $H(\phi_i^*)$ is given by $\underset{v_j}{\operatorname{argmax}} \{ h(v_j) \,|\, \forall v_j \in predecessor(v_e) \cap H(\phi_i^*) \}$.*

Proof. Given $v_a, v_b \in predecessor(v_c)$ with $h(v_a) \geq h(v_b)$, we have $length(\delta_{v_a} \cup v_c) > length(\delta_{v_b} \cup v_c)$. Therefore, the predecessor node of v_e with the latest finish is in the longest path ending with v_e in $H(\phi_i^*)$. □

4.2.3. Explicit Execution Order (ESO)

Tighter boundaries can be achieved by using ESO for NC-joints, because every joint could just be interfered with by concurrent joints with higher priority. Taking the proposed scheduling, a novel analysis method is illustrated, which can sustain CPPE and explicit run sequence of NC-joints.

With joint priority, the interfering joints of v_j on $m-1$ processors could be effectively decreased to joints in $\mathrm{N}(v_j)$ that have a higher priority than p_j, $m-1$ joints in $\mathrm{N}(v_j)$ that have a lower priority and the highest WCET because of the non-preemptive schedule [10]. $\mathrm{N}^e(v_j)$ are the joints that can interfere with a NC-joint v_j with an explicit order, in Formula (3), in which $\mathrm{argmax}_{v_k}^{m-1}$ returns the first $m-1$ joints with the highest value of the given metric. For the sake of simplicity, it takes $(m-1)$ low-priority joints as the upper limit. The better ILP-based method can be used to calculate this congestion accurately. In short, if joint-level preemption is allowed, $\mathrm{N}^e(v_j)$ will further decrease to $\{v_k | q_k > q_j, v_k \in \mathrm{N}(v_j)\}$.

$$\mathrm{N}^e(v_j) = \{v_k | q_k > q_j, v_k \in \mathrm{N}(v_j)\} \cup \underset{v_k}{\overset{m-1}{\mathrm{argmax}}}\{C_k | q_k < q_j, v_k \in \mathrm{N}(v_j)\} \tag{3}$$

$h(v_j), \forall v_j \in V$ could be calculated by Formula (3), $\mathrm{N}^e(v_j)$ applied to NC-joints running on other $m-1$ processors. Therefore, α_i, β_i can be bounded with the updated $h(\phi_i^*)$, $h(v_j), \forall v_j \in H(\phi_i^*) \cup G(\phi_i^*)$. Note that with an explicit schedule, δ_{v_e} calculated in Formula (3), it is not necessarily the longest path in $H(\phi_i^*)$ that runs in the interfering workload [11]. On the contrary, δ_{v_e} offers the path that is always finished last because of the pre-planned joint execution sequence.

However, the final limit of the response time is different from the general situation. With joint priority, whole workload in $(C_i - L_i - \alpha_i - \beta_i)$ do not have to hinder the execution of δ_{v_e}. R^e is the response time of the DAG task with ESO. It is defined in Formula (4), in which decides the joints that could prolong δ_{v_e}, $I_{\delta_{v_e},j}$ offers the real latency on δ_{v_e} from joint v_j in the interfering workload.

$$R^e = \beta_i + \sum\nolimits_{\phi_i^* \in \Phi^*} L_i + \begin{cases} 0 & if \left|\Lambda_{\mathrm{N}^e(\delta_{v_e})}\right| < \mathrm{m} \\ \left\lceil \sum_{v_j \in \mathrm{N}^e(\delta_{v_e})} I_{\delta_{v_e},j} / m \right\rceil & otherwise \end{cases} \tag{4}$$

The length of $\phi_i^*(L_i)$ and the WC latency on $\delta_{v_e}(I_{\delta_{v_e}})$ in the interfering workload, of ϕ_i^* is the WC completion time and $H(\phi_i^*)$ is upper bounded by $\beta_i + L_i + \left\lceil \sum_{v_j \in \mathrm{N}^e(\delta_{v_e})} I_{\delta_{v_e},j} / m \right\rceil$. If the number of paths in the joints that can cause $I_{\delta_{v_e}}$ is smaller than m, $\left(\left|\wedge_{\mathrm{N}^e(\delta_{v_e})}\right| < m\right)$, δ_{v_e} runs directly after ϕ_i^* and finishes by $L_i + \beta_i$. Note that $I_{\delta_{v_e}} = 0$, $\beta_i = 0$, as whole workload in $H(\phi_i^*)$ contributes to α_i so that ϕ_{i+1}^* can start immediately after ϕ_i^*.

These joints can interfere with δ_{v_e} (namely, $\mathrm{N}^e(\delta_{v_e})$) are bound by Formula (5), in which $I_{\delta_{v_e},j}$ offers the real latency from joint v_j on δ_{v_e}.

$$\begin{aligned} \mathrm{N}^e(\delta_{v_e}) = \bigcup_{v_k \in \delta_{v_e}} \{v_j | h(v_j) > h(\phi_i^*) \wedge q_j > q_k, \forall v_j \in \mathrm{N}(v_k)\} \cup \\ \bigcup_{v_k \in \delta_{v_e}} \underset{v_k}{\overset{1 \cdots m}{\mathrm{argmax}}}\{I_{\delta_{v_e},j} | h(v_j) > h(\phi_i^*) \wedge q_j < q_k, v_j \in \mathrm{N}(v_k)\} \end{aligned} \tag{5}$$

$I_{\delta_{v_e},j}$ is given in Formula (6), which takes the workload of v_j executed after $h(\phi_i^*)$ as the WC latency on δ_{v_e}.

$$I_{\delta_{v_e},j} = \begin{cases} C_j, & h(v_j) - C_j \geq h(\phi_i^*) \\ h(v_j) - h(\phi_i^*), & otherwise \end{cases} \tag{6}$$

This concludes the analysis for scheduling methods with node execution order known a priori. As with the general boundary, it is continuable, because the diminution of WCET of any arbitrary joint does not cause the completion later than the WC boundary. This analysis provides stricter results by removing joints that do not cause delay due to their priority than the general limitation of NC-joints with random order, in which $\mathrm{N}^e(v_j) \subseteq \mathrm{N}(v_j)$, $I_{v_e} \leq C_i - L_i - \alpha_i - \beta_i$.

It is noted that the proposed method cannot strictly govern timetable-specific analysis but could offer more accurate results in general. In fact, this threshold can be used as a safety upper limit for recommended analysis to offer the most real approximation of the known worst case.

5. Hierarchical Scheduling in Hypervisor

In order to make full use of the advantages of global scheduling and federal scheduling, we adopt the above new method of scheduling real-time DAG tasks. We use a hierarchical scheduling method to divide the whole scheduling problem into two parts:

(1) Scheduling DAG tasks on virtual processors.
(2) Scheduling virtual processors on physical processors.

More specifically, each DAG task is assigned several dedicated virtual processors (and executed exclusively on these processors at runtime). By correctly describing the resources provided by the virtual processor, we can analyze each DAG task independently as in federated scheduling. On the other hand, virtual processors are scheduled on physical processors at runtime, which effectively enables processors to be shared among different DAG tasks. Therefore, our hierarchical scheduling method inherits the advantages of federal scheduling and global scheduling, thus obtaining better schedulability.

5.1. Overview of Virtualization

The work of this section is different from scheduling tasks to physical cores directly but uses a hierarchical scheduling method to schedule tasks. Furthermore, the hierarchical algorithm schedules every task to the unique virtual processor of this task and then schedules whole virtual cores to the physical core. This method involves an offline design part and an online scheduling part [24,25]. Scheduler-1 schedules the workload of every DAG task to VCPUs. Scheduler-2 schedules the VCPUs on whole the PCPUs. Illustration of Hierarchical scheduling in Figure 4.

5.2. Schedule Tasks to VCPUs

Note that task scheduling is on the virtualization platform. Only the scheduling of a task is considered. Please note that the virtualization environment can only be accessed by its corresponding DAG task. Therefore, when a DAG task is dispatched on its virtual platform Π, there is no interference from other tasks, and the scheduling of every DAG is independent of other tasks [25]. The goal is to construct a virtualization platform π, on which the deadline of k can be arranged by the Scheduler-1 and guaranteed.

5.3. Schedule VCPUs to PCPUs

Note that the problem of scheduling whole VCPUs together on PCPUs Is similar to task scheduling on the virtualization environment. This problem consists of two issues: how to provide specific services required by virtualization environment on PCPUs, and whether these virtual platforms can be successfully scheduled on PCPUs.

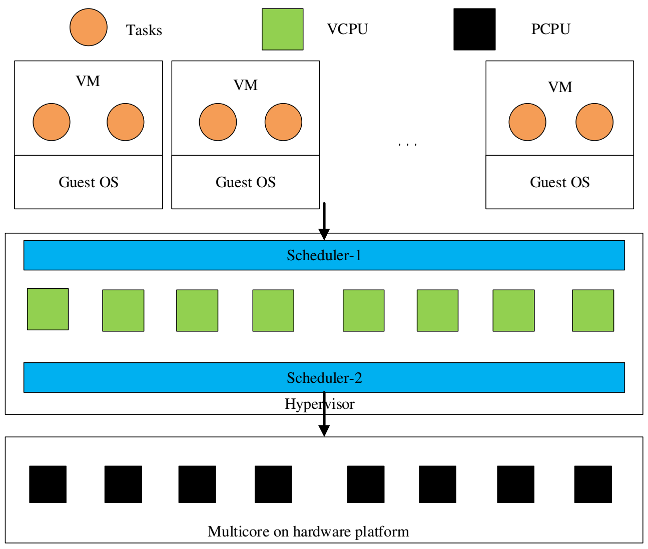

Figure 4. Illustration of Hierarchical scheduling.

5.4. VCPUs in Hypervisor

The architecture of parallel hierarchical scheduling is described in detail. Every task is a DAG, and these tasks cannot directly use processor resources. Instead, processor resources are used in a shared way through middleware. We define middleware as VCPUs. Figure 5 shows the complete architecture of our platform, which is layered. It will implement our approach on avionic system developed based on a hypervisor. Below, it will introduce the implementation of the two scheduling components and the runtime performance in detail. The first problem is how to allocate processor resources to VCPUs. The second is how OpenMP threads use VCPUs. Because the OpenMP thread and system thread are in a one-to-one relationship, using the OpenMP thread instead of the system thread.

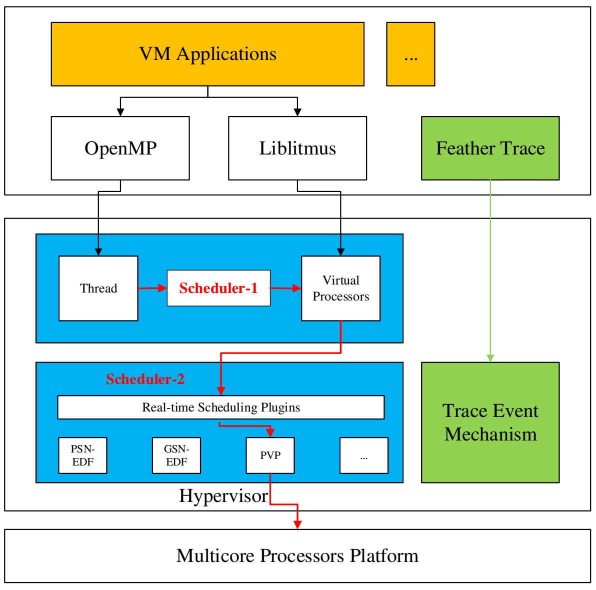

Figure 5. The architecture of virtualization platform.

6. Results

6.1. Evaluations

This experiment verified the capacity proportional to the number of processors. For every configuration, 1000 tests were conducted on the comparison method. Every experiment casually generates a DAG task. The standardized worst-case maximum CT is used as an index. Figure 6 shows the worst-case maximum CT of the existing method and the proposed method under the different numeral of processors on the DAG generated at $p = 8$. When $m \leq 4$, rta-cpf provides similar results to the classical bound, that is, most of its results are the upper bound of the classical bound. This is because the parallelism of DAG is limited when the numeral of cores is small, so every NC-joint has a long worst-case CT (in Formula (3)). This leads to a lower α_i limit (a higher β_i limit) for every supplier, so the worst-case maximum CT is longer. With the further increase of m, rta-cpf becomes effective ($m = 6$), and in the case of $m = 7.8$, it exceeds the classical limit by 15.7% and 16.2% (and as high as 31.7% and 32.2%) on average. In this case, more workloads can be executed in parallel with the CP, that is, the increase of α_i and the decrease of β_i. Therefore, rta-cpf leads to stricter results by explicitly considering this workload, thus safely reducing the interference on the CP. Similar results were obtained in the comparison between rta-cpf-eo and He2019, in which rta-cpf-eo provided a shorter worst-case CT approximation when $m \geq 4$, for example, it was as high as 11.1% and 12.0% when $m = 7$ and $m = 8$, respectively. We note that the execution order of joints in the two methods will also affect the limit of analyzing the worst case. We compare the scheduling and analysis methods respectively. In short, when $m = 7$, rta-cpf provides similar results to it, and is superior to it when $m = 8$. The result shows the usefulness of the proposed method.

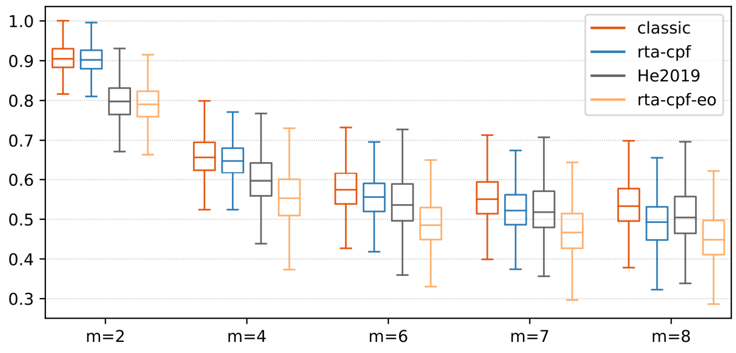

Figure 6. WC CT of DAG using different numeral of processors.

6.1.1. Sensitivity of DAG Priorities

From some results, it is not easy to understand how the DAG attribute affects the WC maximum CT. To adapt to this situation, this experiment shows how the evaluated analysis is sensitive to some DAG features. That is, by controlling the parameters of Dag and evaluating the maximum CT in the normalized value, we can see how much the performance of the analysis has changed. Otherwise, this will not be distinguished by the WC maximum CT or schedulability analysis. Specifically, we consider the following parameters in this experiment: (i) DAG parallelism (the maximum possible width when generating a randomized DAG), p, and (ii) the ratio of DAG CP to total workload $\%L$, where $\%L = L/C \times 100\%$.

Figure 7 evaluates the influence of CP length on the usefulness of the proposed method, $m = 2$. The CP varies from 60% to 90% of the total workload of the generated DAG. In this experiment, compared with the existing methods, the proposed analysis shows the most remarkable performance. For the proposed method, due to the change in the internal structure of the generated DAG (for example, $L = 0.6$), the WC maximum CT of rta-cpf varies with a small amount of $\%L$. However, with the further increase of $\%L$, rta-cpf provides a constant CT because whole non-critical workloads can be executed in parallel with the CP. In this case, the maximum CT is directly equal to the length of the CP. Similar results were obtained for rta-cpf-eo, which provided a constant CT (the length of the CP) under whole experimental settings. Note that, with the further increase of $\%L$, it is completely dominated by rta-cpf. Sensitivity of CP ratio ($m = 4$, $p = 8$) in Figure 8.

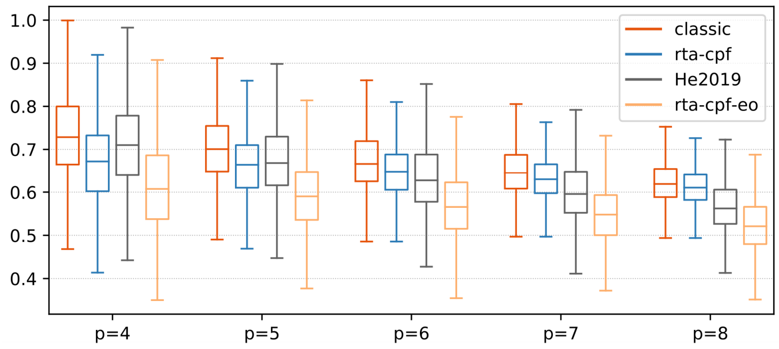

Figure 7. Sensitivity of parallelism parameter ($m = 4$).

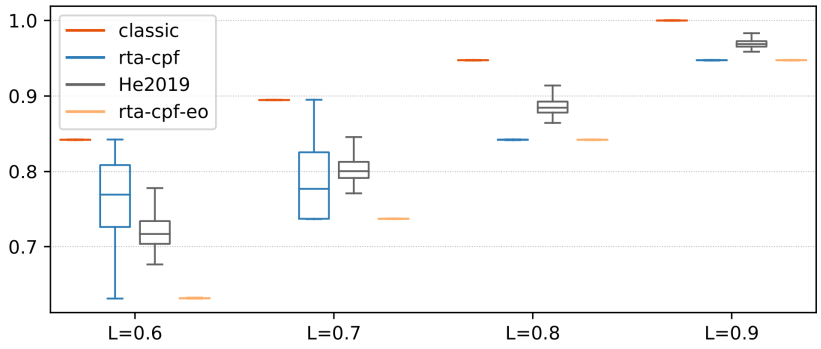

Figure 8. Sensitivity of CP ratio ($m = 4$, $p = 8$).

The method is superior to the classical method and the latest technology in general. In short, we observe that whole the tested parameters have an influence on the function of the proposed method. For rta-cpf, it is sensitive to the relationship between m and p, in which low m or high p shows the usefulness of this method. These two factors have a direct impact on the CT of whole NC-joints. %L will also significantly affect the performance of rta-cpf. In rta-cpf, a longer CP usually leads to a more accurate approximation of the maximum CT. Similar to rta-cpf-eo, rta-cpf shows better performance with the increase of% L. Hence, due to its explicit execution sequence, rta-cpf-eo shows stronger performance than rta-cpf, and is not affected by parameter q.

6.1.2. Usefulness of The Proposed Schedulability

The proposed algorithm is compared with the advanced joint-level priority allocation method (namely He2019). In short, it is proved that the WC achieves maximum CT considering priority allocation. The goal is to prove the WC He2019 of improvement, grouped by the numeral of processors m, $p = 8$, "?" means beat the market. Scenarios are realized through priority allocation. Overall, 1000 random task sets will be generated under every configuration. This evaluation compares two indicators: (i) time suggested rta-cpf-eo is more than the comparison algorithm, (ii) the latency of standardized CT in the improvement cases.

Figure 9 shows the extended comparison between the proposed ranking algorithm and the algorithm in He2019 in the case of different numerals of processors. The frequency denotes the number of cases where the proposed timetable is red or blue. The WC maximum CT analysis for explicit orders is applicable to both orders, so the performance difference comes from the order strategy. From the results, the proposed algorithm is generally superior to it at higher frequencies, especially in the case of a small numeral of cores, for example, in the case of $m = 2$ near the frequency of 600. With the increase of m, the frequency difference of the method gradually decreases, and it becomes difficult to distinguish when $m = 8$. In these cases, most joints can execute in parallel, so the influence of different sequences on the final maximum CT becomes less significant.

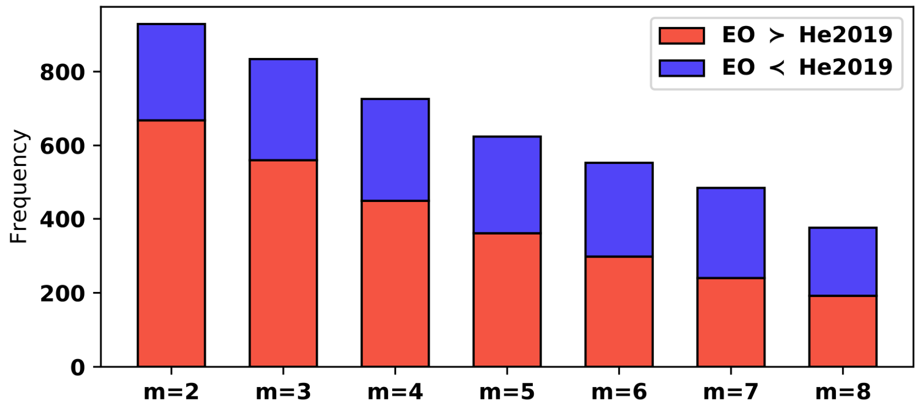

Figure 9. Compare priority sorting with sorting in He2019.

Table 1 compares the advantages of the two algorithms in detail, expressed as a percentage. For EO > He2019 (that is, the proposed schedule is better than it), the average optimization was observed (in terms of the WC maximum CT) is higher than 5.42% (as high as 7.88%) in all cases. In the case of EO < He2019, the optimization is always lower than that of EO > He2019.

Table 1. Advantage: percentage of improvement in joint ordering policy.

He2019 > EO			
Core (*m*)	**Min.**	**Avg.**	**Max.**
2	0.05	7.88	30.62
4	0.02	7.20	33.38
8	0.03	5.42	25.28
He2019 < EO			
Core (*m*)	**Min.**	**Avg.**	**Max.**
2	0.01	6.48	30.67
4	0.02	4.54	23.84
8	0.03	1.64	19.27

Table 2 shows the number of advantageous cases and the scientific significance of improvement, He2019 > EO, He2019 < EO. The values in Table 2 are category values to report its importance. Namely, the importance determines if any difference is more than random, and the size of the difference. The data column illustrates the number of times that an algorithm has a lower CT than another. As far as all of the circumstances are concerned, the algorithm is superior to the most advanced algorithm. This order of magnitude further proves the advantages of our algorithm. For example, when $m = 4$, when EO is superior to He2019, the effect size is medium, but when He2019 is superior to EO, the effect size is small; For $m = 8$, even if data have similar values, the effect size is small and can be ignored.

Table 2. The joint-level priority allocation realized in (α, β).

Core (*m*)	**Data**	**Dataset**
2	262	He2019 > EO
	670	He2019 < EO
4	275	He2019 > EO
	451	He2019 < EO
8	184	He2019 > EO
	191	He2019 < EO

Similarly, by applying the same ranking to the two methods, we compared our analysis with that in He2019, and found consistent results. Therefore, we conclude that the proposed scheduling and analysis is effective and superior to the most advanced technology in general.

6.2. Synthetic Workload

For every DAG task, the number of vertices is casually selected in [50, 250]. The WCET of every vertex is casually selected in [50, 100]. For every possible edge, take a random value in [0, 1], and only add the edge to the graph when the generated value is less than the predefined threshold q. Generally, the larger q, the more ordered the tasks (namely the longer the CP of DAG). In Figure 10. Comparison of acceptance rates of different dimensions under the comprehensive workload [26].

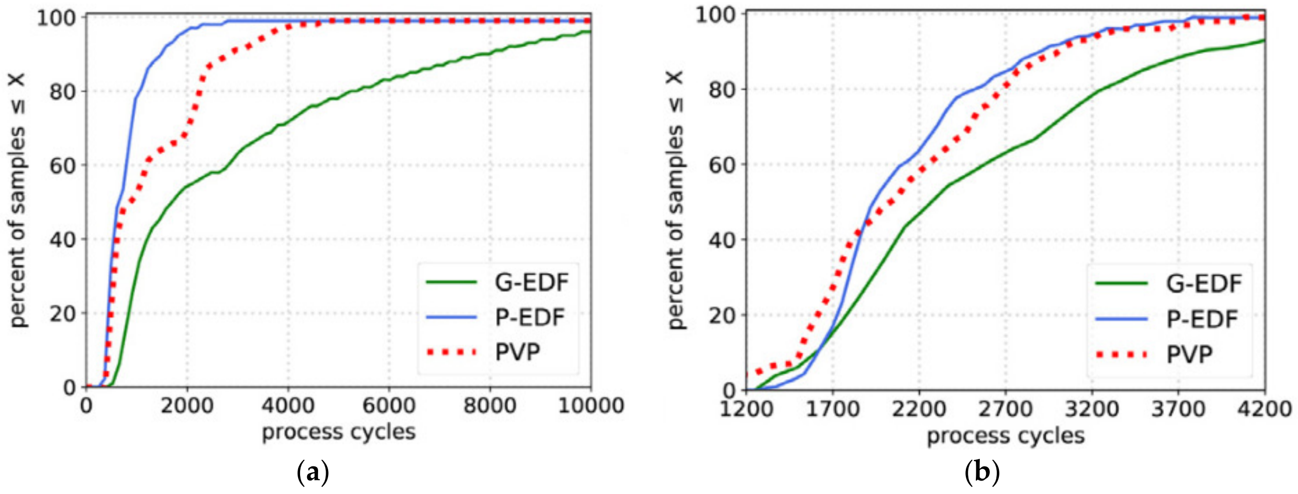

Figure 10. Comparison of three schedulers; (**a**) scheduling overhead, (**b**) context-switch overhead.

Deadline and Period: Periods are generated in an integer power of 2. We find the minimum value A so that $L_i \le c$, and casually set T_i to one of 2c, 2c + 1 or 2c + 2. When the period T_i of the task is 2c, 2c +1 or 2c +2, the ratio L_i/T_i of the task is in the range of [1, 1/2], [1/2, 1/4] or [1/4, 1/8] respectively. The relative cut-off time is uniformly selected from the range $[L_i, T_i]$.

Figure 11a compares the acceptance rates of task sets with different Standardized utilization rates, where q is 0.1 and L_i/T_i is casually generated in the range of [1/8, 1]. Standardized utilization represents the X-axis, and acceptance rates according to the standardized utilization under different processor numerals. It can be observed that SF-XU and H-YANG are superior to G-MEL. This is because SF-XU and H-YANG's analysis techniques are both carried out under the framework of federal scheduling, eliminating the interference between tasks, while G-MEL is based on RTA, which is much more pessimistic due to the interference between tasks. In short, the performance of H-YANG is better than that of SF-XU because it provides a more efficient resource-sharing solution. Figure 11b compares the acceptance rates of task sets with different intensities. Figure 11b follows the same setup as that of Figure 11a, but generates task cycles at different ratios between L_i/T_i. As the x-axis value increases, the tension decreases. The standardized utilization rate of every task is casually selected from [0.1, 1]. Interestingly, with the increase of tension, the gap between these three tests is increasing. It is because, with the increase of the circle in CP length, the problem of resource waste of SF-XU becomes more and more serious, and H-YANG alleviates this problem to some extent.

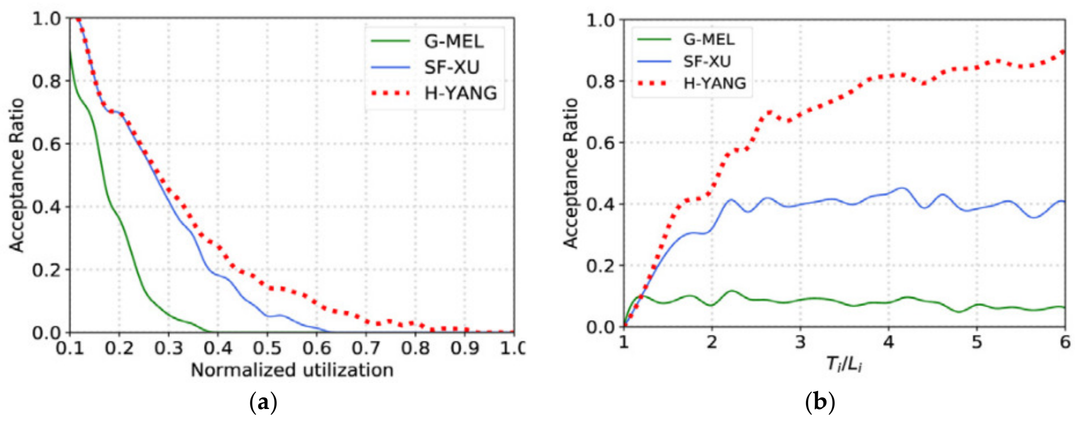

Figure 11. Comparison of different synthetic workload; (**a**) Standardized utilization, (**b**) Tensity.

7. Conclusions

A parallel hierarchical scheduling framework based on DAG tasks with limited deadlines is proposed in this article. Under this framework, the kinds of tasks are not distinguished, and a rule-based scheduling method is proposed which maximizes node parallelism to improve the schedulability of single DAG tasks. Based on the rules, a response-time analysis is developed that provides tighter bounds than existing analysis for (1) any scheduling method that prioritizes the critical path, and (2) scheduling methods with explicit execution order known a priori. We demonstrate that the proposed scheduling and analyzing methods outperform existing techniques. The scheduling framework has been implemented on avionic real-time systems platform, and through experiments, we know that the extra overhead brought by the method proposed in this chapter is acceptable. Finally, a simulation experiment is constructed to verify the adjustability of the framework. The experimental results demonstrate that the strategy proposed have better performance in the article.

Based on the research conclusion of this article, there are still many directions worthy of further analysis in the future. First, we can consider scheduling multi-DAG task models on a given virtual computing platform according to different resource interfaces. Moreover, sharing resource models among DAG tasks is also an important direction for us to consider. Energy-sensitive scheduling algorithm for the DAG task model is also worthy of attention.

Author Contributions: Conceptualization, H.Y. and S.Z. (Shuang Zhang); methodology, H.Y. and S.Z. (Shuai Zhao); software, H.Y. and S.Z. (Shuai Zhao); validation, Y.G., H.Y. and X.S.; formal analysis, Y.G.; investigation, H.Y.; resources, S.Z. (Shuang Zhang); data curation, X.S.; writing—original draft preparation, H.Y.; writing—review and editing, Y.G.; visualization, X.S.; supervision, S.Z. (Shuang Zhang); project administration, S.Z. (Shuang Zhang); funding acquisition, H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Key Scientific Research Projects of China, grant numeral MJ-2018-S-34.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The work was supported by the school of computer at Northwestern Polytechnical University, some colleagues in University of York and Xi'an Aeronautical Computing Technique Research Institute.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alonso, A.; de la Puente, J.; Zamorano, J.; de Miguel, M.; Salazar, E.; Garrido, J. Safety concept for a mixed criticality on-board software system. *IFAC PapersOnLine* **2015**, *48*, 240–245. [CrossRef]
2. Burns, A.; Davis, R. A survey of research into mixed criticality systems. *ACM Calc. Surv.* **2017**, *50*, 1–37. [CrossRef]
3. Burns, A.; Davis, R. *Mixed Criticality Systems: A Review*; Technical Report MCC-1(L); Department of Calculater Science, University of York: York, UK, 2019; p. 6. Available online: http://www-users.cs.york.ac.uk/burns/review.pdf (accessed on 10 March 2019).
4. LynxSecure. Available online: https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor (accessed on 14 January 2021).
5. QNX Adaptive Partitioning Thread Scheduler. Available online: https://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.sys_arch/topic/adaptive.html (accessed on 14 January 2021).
6. QNX Hypervisor. Available online: https://blackberry.qnx.com/en/software-solutions/embedded-software/industrial/qnx-hypervisor (accessed on 14 January 2021).
7. QNX Platform for Digital Cockpits. Available online: https://blackberry.qnx.com/content/dam/qnx/products/bts-digital-cockpits-product-brief.pdf (accessed on 14 January 2021).
8. Wind River Helix Virtualization Platform. Available online: https://www.windriver.com/products/helix-platform/ (accessed on 18 June 2021).
9. Wind River VxWorks 653 Platform. Available online: https://www.windriver.com/products/vxworks/certification-profiles/#vxworks_653 (accessed on 18 June 2019).
10. Baruah, S. The Federated Scheduling of Systems of Conditional Sporadic DAG Tasks. In Proceedings of the 12th International Conference on Embedded Software, Amsterdam, The Netherlands, 4–9 October 2015; pp. 1–10.
11. Li, J.; Chen, J.; Agrawal, K.; Lu, C.; Gill, C.; Saifullah, A. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In Proceedings of the 26th Euromicro Conference on Real-Time Systems, Madrid, Spain, 8–11 July 2014; pp. 85–96.
12. Xu, J.; Nan, G.; Xiang, L.; Wang, Y. Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors. In Proceedings of the 2017 IEEE Real-Time Systems Symposium (RTSS), Paris, France, 5–8 December 2017; pp. 80–91.
13. He, Q.; Jiang, X.; Guan, N.; Guo, Z. Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 2283–2295. [CrossRef]
14. Melani, A.; Bertogna, M.; Bonifaci, V.; Marchetti-Spaccamela, A.; Buttazzo, G.C. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In Proceedings of the Euromicro Conference on Real-Time Systems, Lund, Sweden, 8–10 July 2015; pp. 211–221.
15. Graham, R.L. Bounds on multiprocessing timing anomalies. *J. Appl. Math.* **1969**, *17*, 416–429. [CrossRef]
16. Fonseca, J.; Nelissen, G.; Nélis, V. Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling. In Proceedings of the International Conference on Real-Time Networks and Systems, Grenoble, France, 4–6 October 2017; pp. 28–37.
17. Chen, P.; Liu, W.; Jiang, X.; He, Q.; Guan, N. Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems. *ACM Trans. Embed. Comput. Syst.* **2019**, *18*, 1–19. [CrossRef]
18. Chang, S.; Zhao, X.; Liu, Z.; Deng, Q. Real-time scheduling and analysis of parallel tasks on heterogeneous multi-cores. *J. Syst. Archit.* **2020**, *105*, 101704. [CrossRef]
19. Guan, F.; Qiao, J.; Han, Y. DAG-fluid: A real-time scheduling algorithm for DAGs. *IEEE Trans. Calc.* **2020**, *70*, 471–482. [CrossRef]
20. Topcuoglu, H.; Hariri, S.; Wu, M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.* **2002**, *13*, 260–274. [CrossRef]
21. Lin, H.; Li, M.-F.; Jia, C.-F.; Liu, J.-N.; An, H. Degree-of-joint task scheduling of fine-grained parallel programs on heterogeneous systems. *J. Calc. Sci. Technol.* **2019**, *34*, 1096–1108.
22. Zhao, S.; Dai, X.; Bate, I.; Burns, A.; Chang, W. DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency. In Proceedings of the 2020 IEEE Real-Time Systems Symposium (RTSS), Houston, TX, USA, 1–4 December 2020; pp. 28–40.
23. Zhao, S.; Dai, X.; Bate, I. DAG Scheduling and Analysis on Multi-Core Systems by Modelling Parallelism and Dependency. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 231–245. [CrossRef]
24. Jiang, X.; Guan, N.; Long, X.; Wan, H. Decomposition-based Real-Time Scheduling of Parallel Tasks on Multi-cores Platforms. *IEEE Trans. Calc. Aided Des. Integr. Circuits Syst.* **2019**, *39*, 183–198.
25. Yang, T.; Deng, Q.; Sun, L. Building real-time parallel task systems on multi-cores: A hierarchical scheduling approach. *J. Syst. Archit.* **2019**, *92*, 1–11. [CrossRef]
26. Saifullah, A.; Ferry, D.; Li, J.; Agrawal, K.; Lu, C.; Gill, C. Parallel real-time scheduling of dags. *Parallel Distrib. Syst. IEEE Trans.* **2014**, *25*, 3242–3252. [CrossRef]