Real-Time Systems (2024) 60:291–327
https://doi.org/10.1007/s11241-024-09421-9

Real-time scheduling for parallel tasks with resource reclamation

Qingqiang He[1] · **Yongzheng Sun**[1] · **Xu Jiang**[2] · **Mingsong Lv**[1,2] · **Jinkyu Lee**[3] · **Nan Guan**[4]

Accepted: 6 May 2024 / Published online: 6 June 2024
© The Author(s) 2024

Abstract

This paper considers the real-time scheduling of a parallel task with reclaiming computing resources, which can be utilized for soft real-time tasks or switching to low-energy mode to save energy. Existing works allocate a *rectangular* piece of computing resources based on the worst-case characterizations of the task to guarantee the deadline, which inherently incurs severe resource wasting due to coarse-grained resource allocation. To address this resource-wasting problem, this paper proposes the *ladder-like* resource allocation (i.e., a series of rectangular pieces of computing resources). To characterize the ladder-like resource allocation, we present two concepts called *resource distribution* and *allocation vector*, which serve as the interfaces between hard and soft real-time tasks. For the former, we derive schedulability tests under the given two interfaces; for the latter, we discuss the methods of determining the two interfaces to reclaim computing resources. This paper is the first work to fully explore the concept of ladder-like resource allocation and its potential consequences on computing resources, soft real-time tasks, and energy. Experiments demonstrate that the proposed approach can effectively reclaim more computing resources than existing approaches while maintaining hard real-time guarantees.

Keywords Real-time scheduling · Parallel task · Resource reclamation · Response time analysis

1 Introduction

Real-time scheduling theory focuses on the analysis of tasks to meet hard real-time guarantees for time-critical embedded systems. In time-critical embedded systems, such as autonomous driving (Sun et al. 2023), robotics (Li et al. 2022) and satellite communication (Lv et al. 2022), the correctness of tasks does not only depend on the logical results, but also the time when the results are produced. For tasks in these systems, a single deadline miss may cause catastrophic consequences. Therefore,

Extended author information available on the last page of the article

before the deployment of time-critical embedded systems, real-time analysis is required to formally verify that all timing constraints are guaranteed under any circumstance. As multi-core platforms are increasingly used in real-time systems for performance and energy efficiency, most applications are parallelized to take advantage of the power of multi-cores. Real-time analysis of parallel tasks has gained much attention in recent years (Li et al. 2013; Baruah 2015a; Ueter et al. 2018; Chen et al. 2019; Jiang et al. 2021; He et al. 2022).

Among the real-time scheduling approaches for parallel tasks, federated scheduling (Li et al. 2014) is a promising approach with guaranteed real-time performance. In federated scheduling,[1] each parallel task is allocated a rectangular piece of computing resources where a fixed number of cores are available in a time interval of length equal to the deadline (see Fig. 7). This rectangular resource allocation suffers from the resource-wasting problem (Jiang et al. 2021) due to the pessimism within its analysis techniques and the overly conservative characterizations of a parallel real-time task. For example, a typical characterization of a parallel real-time task is the worst-case execution time. Federated scheduling allocates computing resources based on this worst-case execution time. However, the worst-case execution time is very conservative and pessimistic. During executions, the actual execution time may be far less than the worst-case execution time of tasks, resulting in that many computing resources are wasted. To address the resource-wasting problem, this paper proposes the *ladder-like resource allocation*, which consists of a series of rectangular pieces of computing resources (see Fig. 3). First, the ladder-like resource allocation is able to roughly capture the "shape" of executions of the task (i.e., how many cores are used in different time intervals during executions) through offline profiling, thus reclaiming the unused computing resources. Second, the ladder-like resource allocation can support the dynamic adjustment of cores allocated to the parallel task through online monitoring the execution, thus further reclaiming computing resources. To characterize the ladder-like resource allocation, we present two concepts called *resource distribution* and *allocation vector*. Intuitively, the resource distribution describes how computing resources are assigned between the time at which the task starts execution and the deadline. The allocation vector describes at which time the number of cores should be adjusted based on the online monitored information. The resource distribution and allocation vector serve as the interfaces between hard and soft real-time tasks. For hard real-time tasks, schedulability tests under the given two interfaces are derived; for soft real-time tasks, the methods of determining the two interfaces to reclaim computing resources are presented. To our best knowledge, this paper is the first work to fully explore the concept of ladder-like resource allocation and its potential consequences on computing resources, soft real-time tasks, and energy saving.

Being able to guarantee the deadline for the hard real-time task, our approach can reclaim computing resources for soft real-time tasks or switch the reclaimed

[1] In federated scheduling, there are light tasks and heavy tasks. Since light tasks are treated as sequential tasks and heavy tasks are executed in isolation, this paper focuses on the scheduling of a single heavy task.

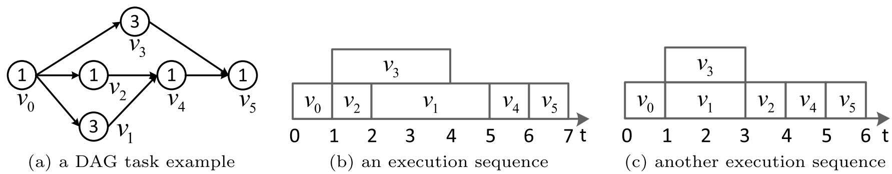


Fig. 1 An illustrative example

computing resources to low-energy mode for saving energy. Note that this paper does not provide the analysis or techniques for soft-real time tasks or energy. Since both soft real-time tasks and energy are related to the amount of reclaimed computing resources from hard real-time tasks, in the paper, we take the target of reclaiming computing resources for soft real-time tasks as a use case to present our methods. The proposed approach only relies on the volume (i.e., the total workload) and the length (i.e., the length of the longest path) of the parallel real-time task, not requiring the detailed structure of the task. Experiments through simulating randomly generated parallel tasks demonstrate that the proposed approach outperforms existing approaches by 48.3% at the maximum and by 37.8% on average regarding reclaimed computing resources when maintaining hard real-time guarantees at the same time.

Organization. Section 2 reviews related works. Section 3 defines the task model and the scheduling algorithm. Section 4 discusses the pessimism in existing approaches which motivates this work. Sections 5 and 6 propose two techniques for reclaiming computing resources. Section 7 presents the proposed approach using these two resource-reclaiming techniques. Evaluation results are reported in Sect. 8, and Sect. 9 concludes the paper.

2 Related work

For real-time scheduling of parallel tasks, there are lots of works where parallel real-time tasks are represented as the gang task model (Dong and Liu 2019; Lee et al. 2022), the fork-join task model (Lakshmanan et al. 2010), or the DAG-based task model (Li et al. 2013; Melani et al. 2015; Sun et al. 2017; Liang et al. 2023). The gang task model describes the simple parallel execution behaviors and requires that different parallel subtasks must execute on several cores simultaneously (Dong and Liu 2019; Lee et al. 2022). The fork-join task model describes structures with interleaving sequential and parallel subtasks. This type of structure is common in parallel applications such as OpenMP (Lakshmanan et al. 2010). The DAG-based task model represents the parallel subtasks as a directed acyclic graph (Li et al. 2013) and an example of DAG tasks is illustrated in Fig. 1. See (Tang et al. 2022) for a comprehensive introduction of the task models for parallel real-time tasks.

The scheduling methods for parallel real-time tasks can be categorized into four major paradigms: decomposed scheduling (Qamhieh et al. 2013; Jiang et al. 2016), partitioned scheduling (Fonseca et al. 2016; Casini et al. 2018), global scheduling (Li et al. 2013; He et al. 2021, 2023b), and federated scheduling (Li et al. 2014).

Among the four scheduling paradigms, federated paradigm provides the best performance regarding schedulability and is simple to implement. In federated paradigms, each parallel real-time task is scheduled independently on a set of dedicated cores. Federated scheduling in (Li et al. 2014) was generalized to constrained deadline tasks (Baruah 2015a), arbitrary deadline tasks (Baruah 2015b), and tasks with conditional branches (Baruah 2015c). A series of federated-based scheduling approaches (Jiang et al. 2017; Ueter et al. 2018; Jiang et al. 2020, 2021; He et al. 2022, 2023a) were proposed to address the resource-wasting problem in the original federated scheduling. Although the above methods improve the offline analysis to mitigate the resource-wasting, all of these methods are based on the worst-case execution time of parallel tasks, which means that the computing resources have to be reserved under worst-case assumptions. However, as observed in (Baruah 2018), it may be the case that most executions of the task will have resource demand far below the worst-case assumptions, which means severe resource-wasting. This type of resource-wasting cannot be addressed by the above-mentioned methods.

This paper follows a different line of research to tackle the resource-wasting problem, which combines offline analysis and online monitoring to adjust the allocated computing resources dynamically, thus reclaiming resources for soft real-time tasks or for saving energy. There exist two closely related works (Agrawal and Baruah 2018; Baruah 2018). Agrawal and Baruah (2018) originally proposed an approach of dynamically changing the number of cores allocated to a parallel real-time task to achieve resource-efficient executions. In the study, a task model was proposed to represent parallel real-time tasks using two pairs of volume and length with different levels of assurance. This task model enables the scheduling algorithm to dynamically adjust the number of cores assigned to an individual task during runtime. Baruah (2018) extended the method in (Agrawal and Baruah 2018) by combining the worst-case characterizations (i.e., the volume and length) and offline profiling the execution behavior of parallel real-time tasks. The study was motivated using conditional parallel tasks. The existence of conditional structures makes the execution behavior of parallel real-time tasks more complex and the worst-case characterizations more pessimistic, which exacerbates the resource-wasting problem. While the former (Agrawal and Baruah 2018) provides the analysis method to ensure that the hard real-time task meets its deadline if the allocated number of cores is dynamically adjusted, the latter (Baruah 2018) provides a concrete method of adjusting the allocated number of cores dynamically.

Again, we want to highlight that our work is fundamentally different from the federated-based scheduling approaches (Li et al. 2014; Jiang et al. 2017; Ueter et al. 2018; Jiang et al. 2020, 2021; He et al. 2022, 2023a), particularly the semi-federated scheduling (Jiang et al. 2017) and the virtually-federated scheduling (Jiang et al. 2021). All of these works decide the computing resource allocation in offline analysis based on the worst-case execution time of tasks. However, the worst-case execution time can be rather pessimistic and the actual execution time can be far less than the worst-case execution time, which leads to significant resource-wasting. In contrast, our work combines offline analysis and online monitoring to reclaim computing resources for soft real-time tasks or for saving energy. A simple example can be used to illustrate this. There are two cores and two tasks. For both tasks, the

period and deadline are the same and are 10. The worst-case execution times of both tasks are also 10. In this simplified example, for the federated scheduling (Li et al. 2014), the semi-federated scheduling (Jiang et al. 2017) and the virtually-federated scheduling (Jiang et al. 2021), the resource allocations are the same: the first task is scheduled on the first core; the second task is scheduled on the second core. During offline analysis, no computing resources are wasted, since each task requires 100% computing resources of their allocated core to ensure the hard real-time guarantees. However, during online execution, suppose the actual execution times of the two tasks are both 1. In this case, 90% of computing resources are wasted during execution; no offline analysis methods based on the worst-case execution time can reclaim this 90% computing resources. In contrast, methods with online monitoring, such as (Agrawal and Baruah 2018; Baruah 2018) and our method, can potentially reclaim this 90% computing resources depending on the respective techniques and design choices.

3 System model

This section presents the system model. Section 3.1 defines the DAG task model, which is used in the analysis for deriving the proposed approach. Section 3.2 describes the runtime behavior of the DAG task model defined in Sect. 3.1. Section 3.3 clarifies the task model that is needed for implementing the proposed approach. Some information that is needed for derivation is not required for implementation.

3.1 Task model for the analysis

The sporadic hard real-time task is specified as a tuple (G, D, T), where G is the DAG task model, D is the relative deadline and T is the period. We consider constrained deadlines, i.e., $D \leq T$. The DAG task model is a directed acyclic graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. A vertex $v \in V$ represents a piece of sequentially executed code with a worst-case execution time (WCET) $c(v)$. An edge $(v_i, v_j) \in E$ means that v_j can only start its execution after v_i completes its execution. A vertex with no incoming edges is called a *source vertex* and a vertex with no outgoing edges is called a *sink vertex*. Without loss of generality, we assume that G has exactly one source (denoted as v_{src}) and one sink (denoted as v_{snk}). If G has multiple source or sink vertices, we add a source or sink vertex with zero WCET to comply with the assumption.

A *path* λ is a set of vertices $(\pi_0, \cdots, \pi_k)$ such that $\forall i \in [0, k-1]$: $(\pi_i, \pi_{i+1}) \in E$. The length of a path λ is defined as $len(\lambda) := \sum_{\pi_i \in \lambda} c(\pi_i)$. A *complete path* is a path $(\pi_0, \cdots, \pi_k)$ such that $\pi_0 = v_{src}$ and $\pi_k = v_{snk}$. In other words, a complete path is a path starting from the source vertex and ending at the sink vertex. The *longest path* is a complete path with the largest length among all paths in the task. If there is an edge $(u, v) \in E$, u is a *predecessor* of v. We use $pred(v)$ to denote the set of predecessors of v.

Example 1 Fig. 1a shows a parallel real-time task G where the number inside vertices represents the WCET. The relative deadline and the period $D = T = 7$. v_0 and v_5 are the source vertex and the sink vertex, respectively. The longest path is $\lambda = (v_0, v_1, v_4, v_5)$, so $len(\lambda) = 6$. For vertex v_5, $pred(v_5) = \{v_3, v_4\}$.

3.2 Runtime behavior

The hard real-time task G executes on a multi-core platform with m identical cores. A vertex is said to be *eligible* if all of its predecessors have finished their executions. In other words, an eligible vertex can be immediately executed if there are available cores. In *work-conserving* scheduling, an eligible vertex must be executed if there are available cores. The parallel task G is scheduled by a work-conserving scheduler.

At runtime, vertices of G execute at certain time points on certain cores under the decision of the scheduling algorithm. An *execution sequence* ε of G describes which vertex executes on which core at every time point. For a vertex v, the *start time* $s(v)$ and *finish time* $f(v)$ are the time point when v first starts its execution and finishes its execution, respectively. Note that $s(v)$ and $f(v)$ are related to a certain execution sequence ε. Here we do not include ε in the notations for simplicity. Without loss of generality, we assume the task G starts execution at time 0, so the *response time* R of G in an execution sequence equals the finish time of the sink vertex, i.e., $f(v_{snk})$.

Example 2 For the DAG G in Fig. 1a, suppose the number of cores $m = 2$. Figure 1b and c show two possible execution sequences under work-conserving scheduling. In Fig. 1b, every vertex in G executes for its WCET. In Fig. 1c, v_1 and v_3 execute for less than their WCETs. In Fig. 1c, the start time and finish time of v_3 are $s(v_3) = 1$ and $f(v_3) = 3$, respectively. The response times of G for execution sequences in Fig. 1b and c are 7 and 6, respectively. For the execution in Fig. 1b, v_2 and v_3 execute in parallel; part of v_1 and v_3 execute in parallel. For the execution in Fig. 1c, v_1 and v_3 execute in parallel.

3.3 Task model for the proposed approach

The proposed approach only uses two characterizations (i.e., volume and length) of the hard real-time task, not requiring the internal graph structure of the task. The DAG task model only serves as an auxiliary tool for deriving the schedulability test, not used in the schedulability test itself. The *volume* of a parallel task G (denoted as $vol(G)$) is the total workload in this task. The *length* of a parallel task G (denoted as $len(G)$) is the length of the longest path in this task. The volume can be measured by executing the task in a platform with one core, and the length can be measured by executing the task on a platform with a sufficiently large number (bounded by the number of vertices in this task) of cores (Agrawal and Baruah 2018).

For the DAG task model introduced in Sect. 3.1, the volume $vol(G) = \sum_{v \in V} c(v)$. As an example, for the DAG in Fig. 1a, the volume $vol(G) = 10$ and the length $len(G) = 6$. Besides the DAG task model, the proposed approach is also applicable to other graph

task models, such as the conditional DAG task model (Baruah et al. 2015; Melani et al. 2015; He et al. 2023c), the task model with non-well-nested conditional structures (Sun et al. 2020), and the task model with loop structures (Sun et al. 2021).

For graph task models, an *execution flow* $\mathcal{F}$ is a subgraph that contains all the vertices and related edges during an execution (Sun et al. 2020). In other words, an execution flow is a runtime instance of a task graph. A graph task model can be applied in the proposed approach, if it satisfies all of the following conditions.

1. The execution flows of the graph task model are DAG tasks defined in Sect. 3.1.
2. The runtime behavior of the execution flows conforms to Sect. 3.2.
3. The volume and length of the graph task model can be computed or measured.

Note that graph task models that take into account resource contention (Bi et al. 2022), communication cost between vertices (Chen et al. 2020), and heterogenous computing platforms (Han et al. 2019; Voudouris et al. 2022), violate Conditions 1 and 2, thus not applicable to the proposed approach.

4 Motivation

This section discusses the scheduling algorithms for a parallel real-time task in federated scheduling (Li et al. 2014; Baruah 2015d), which motivates this work. Both DAG tasks and conditional DAG tasks are discussed concerning the resource-wasting problem.

4.1 Discussion on DAG tasks

We first discuss scheduling algorithms for the parallel application modeled as a DAG task in Sect. 3.1. In (Graham 1969), Graham proposed a well-known response time bound using the volume and the length of a DAG task as follows. The response time R of DAG task G scheduled by work-conserving scheduling on m cores is bounded by Eq. (1).

$$R \leq len(G) + \frac{vol(G) - len(G)}{m} \tag{1}$$

Therefore, in federated scheduling, the number of cores m allocated to DAG task G can be computed by Eq. (2).

$$m = \left\lceil \frac{vol(G) - len(G)}{D - len(G)} \right\rceil \tag{2}$$

Equation (2) computes the minimum number of cores m such that the response time bound in Eq. (1) is no larger than the deadline D.

For a DAG task, the computing resources allocated according to Eqs. (1) and (2) exhibit several types of pessimism and cause a large amount of resources being wasted, as summarized in the following.

- **Analysis Pessimism (Type-1).** The bound in Eq. (1) is derived by constructing an artificial scenario where vertices not in the longest path do not execute in parallel with the execution of the longest path (He et al. 2022). However, in real execution, many vertices not in the longest path actually can execute in parallel with the longest path. As observed in (Jiang et al. 2021), this type of pessimism may cause the portion of wasted computing resources to be arbitrarily close to 100%.
- **Execution Pessimism (Type-2).** Parameters used in Eq. (2), such as $vol(G)$ and $len(G)$, are based on the worst-case execution time. To comply with the hard real-time requirements, these worst-case execution times can be overly pessimistic (Edgar and Burns 2001; Bernat et al. 2002), and the actual execution time can be far less than the WCET, leading to severe resource-wasting during execution.

4.2 Discussion on conditional DAG tasks

For parallel tasks modeled as conditional DAG tasks (He et al. 2023c), new types of pessimism appear and the resource-wasting problem is exacerbated. To illustrate this, we first discuss the analysis and scheduling of conditional DAG tasks in federated scheduling (Baruah 2015d).

An execution flow of conditional DAG tasks is a DAG, so notations for DAG tasks in Sect. 3 can also be applied to execution flows. For a conditional DAG task G, we also use G to denote the set of execution flows that this conditional DAG task may generate during executions. A conditional DAG task may generate an exponential number of execution flows. An execution flow, which is a DAG task, may correspond to many execution sequences (see Example 2).

According to Eq. (1), it can be easily seen that the response time R of conditional DAG task G scheduled by work-conserving scheduling on m cores is bounded by Eq. (3).

$$R \leq \max_{\mathcal{F} \in G} \left\{ len(\mathcal{F}) + \frac{vol(\mathcal{F}) - len(\mathcal{F})}{m} \right\} \tag{3}$$

For a conditional DAG task G, the volume is

$$vol(G) = \max_{\mathcal{F} \in G} \{vol(\mathcal{F})\} \tag{4}$$

And the length is

$$len(G) = \max_{\mathcal{F} \in G} \{len(\mathcal{F})\} \tag{5}$$

The volume and length can be computed in polynomial time in the representation of the conditional DAG task (Baruah et al. 2015; Melani et al. 2015). Same as DAG tasks, the volume and length of conditional DAG tasks can also be measured without knowing the detailed graph structure of the task.

It is shown in (Baruah 2015d) that the response time R of conditional DAG task G scheduled by work-conserving scheduling on m cores is bounded by Eq. (6).

$$R \leq len(G) + \frac{vol(G) - len(G)}{m} \tag{6}$$

The bound in Eq. (6) is an approximation of Eq. (3) and always equal to or larger than the bound in Eq. (3). Same as Eq. (2), in federated scheduling, the number of cores m allocated to a conditional DAG task G can be computed by Eq. (7).

$$m = \left\lceil \frac{vol(G) - len(G)}{D - len(G)} \right\rceil \tag{7}$$

Since a DAG task is a special case of a conditional DAG task, the two types of pessimism for DAG tasks also exist in the scheduling of conditional DAG tasks. For a conditional DAG task, there are two additional types of pessimism in the computing resources allocated according to Eqs. (6) and (7), as summarized in the following.

- **Analysis Pessimism (Type-3).** In Eq. (6), the volume and length may correspond to different branches of a conditional construct. Different branches of a conditional construct are mutually exclusive which means that in one execution flow, two branches of a conditional construct cannot both exist (Baruah 2015d). Therefore, the bound in Eq. (6) includes scenarios that cannot actually happen during execution, which leads to the over-provisioning of computing resources.
- **Execution Pessimism (Type-4).** During execution, the entry vertices of a conditional construct can be evaluated to different values, thus leading to different branches and generating different execution flows. These execution flows can have quite diversified characterizations, such as volume, length and the degree of parallelism (He et al. 2023a), thus requiring quite different computing resources. However, the computing resources must be allocated according to the worst-case scenario as shown in Eq. (3) to ensure that the deadline is satisfied (Baruah 2018).

Among these four types of pessimism for parallel real-time tasks, the analysis pessimism (i.e., Type-1 and Type-3) can be partially addressed through improved offline analysis. For example, Type-1 pessimism can be partially addressed by the technique of intra-task priority assignment (He et al. 2019; Zhao et al. 2020; He et al. 2021) or the technique of long paths (He et al. 2022, 2023a). Type-3 pessimism can be eliminated by directly computing the bound in Eq. (3) and utilizing an iterative procedure to find the minimum number of cores such that the bound in Eq. (3) is no larger than the deadline with the sacrifice of time complexity (Melani et al. 2016). However, the execution pessimism (i.e., Type-2 and Type-4) cannot be mitigated through offline analysis, since the required information (such as the actual execution time of vertices, or which branch of a conditional structure is taken) is only available during runtime.

4.3 Overview of the proposed approach

In this paper, we propose an approach for a parallel real-time task to address all four types of pessimism with the target of both satisfying hard real-time deadlines and reclaiming computing resources for soft real-time tasks. The proposed approach utilizes two resource-reclaiming techniques. The execution of the task is divided into several slots. These slots are ladder-like resource allocations rather than a single rectangular allocation. Section 5 presents the first technique by offline profiling the execution of the hard real-time task to reclaim computing resources in the beginning slots. Section 6 presents the second technique by online monitoring the execution of the hard real-time task to reclaim computing resources in the last slot. The proposed approach is presented in Sect. 7, which jointly employs these two techniques to reclaim computing resources.

5 Reclaiming resources in the beginning

This section introduces the technique to reclaim computing resources in the beginning part of executions by offline profiling the workload during various executions of the parallel real-time task. The computing resources are allocated according to the profiled information.

5.1 Intuition of the technique

Due to the complex structure of parallel tasks, different parts of a parallel task may have different numbers of vertices that can execute in parallel. Therefore, during an execution, the used number of cores can vary from time to time. If the allocated number of cores is constant and thus the allocated computing resources are a rectangle as in federated scheduling, a large amount of computing resources can be wasted. In this section, instead of the rectangular resource allocation, we use the ladder-like resource allocation, where different numbers of cores are allocated in different time intervals (see Fig. 3).

Aside from reclaiming computing resources for soft real-time tasks, the ladder-like resource allocation is also capable of reducing the computing resources for hard real-time tasks. In federated scheduling, by Eqs. (2) and (7), if the required number of cores before applying the ceiling function is $m + \epsilon$ ($0 < \epsilon < 1$), then the allocated number of cores has to be $m + 1$, which leads to resource-wasting. This issue is observed and addressed in (Jiang et al. 2017) through relatively complicated analysis. However, we observe that this problem can be easily solved by using the ladder-like resource allocation as illustrated in Example 3.

Example 3 A parallel real-time task G has volume $vol(G) = 26$, length $len(G) = 5$, and deadline $D = 15$. Since $\frac{vol(G)-len(G)}{D-len(G)} = 2.1$, by Eq. (2), the allocated number of cores is $m = 3$, and the allocated computing resources are $m \times D = 45$ as shown in

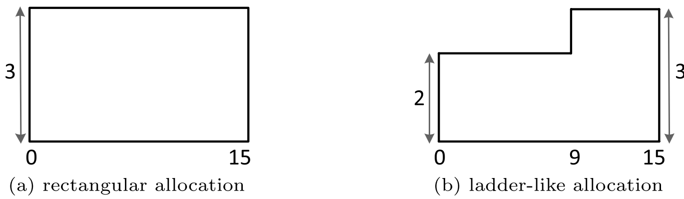


Fig. 2 An example explaining the difference between rectangular and ladder-like resource allocations

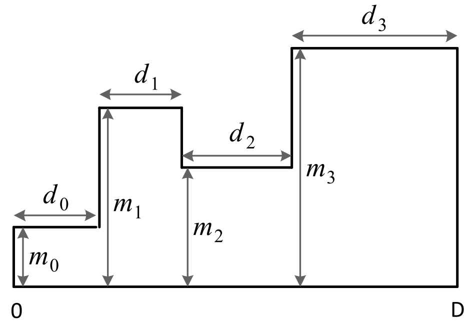


Fig. 3 Illustration of the scheduling for the technique in Sect. 5

Fig. 2a. Under the ladder-like resource allocation, the allocated computing resources can be 2 cores in time interval [0, 9], and 3 cores in time interval [9, 15] as shown in Fig. 2b (the reason why this resource allocation can guarantee the deadline will be explained in Sect. 5.3). And the allocated computing resources are $2 \times 9 + 3 \times 6 = 36$.

Deciding the resource distribution lies in the center of the proposed technique in this section. As stated before, the purposes of this paper are to, first guarantee the deadline for hard real-time tasks, and second reclaim computing resources for soft real-time tasks. In the following, Sect. 5.2 formally defines the resource distribution. Section 5.3 derives the conditions that the resource distribution should satisfy to guarantee the deadlines for hard real-time tasks. Section 5.4 discusses the heuristics for designing the resource distribution such that the reclaimed computing resources for soft real-time tasks can be as many as possible. The final resource distribution for this technique should, first satisfy the conditions derived in Sect. 5.3, and second comply with the heuristics discussed in Sect. 5.4. Section 5.5 provides an example to explain the proposed technique.

5.2 The scheduling algorithm

This subsection describes the computing resources allocated for the proposed technique and the scheduling on the computing resources. The allocated computing resources are characterized by a concept called *resource distribution*.

Definition 1 (Resource Distribution) The resource distribution Θ is defined as $\{(m_0, d_0), \cdots, (m_k, d_k)\}$ $(k \geq 0)$ where (m_i, d_i) means that the available number of cores during a time interval with length d_i is m_i. Θ satisfies all of the following conditions.

1. $\forall i \in [0, k]$: $m_i \geq 1$ and $d_i > 0$.
2. $\forall i, j \in [0, k]$ and $i < j$: (m_i, d_i) is allocated before (m_j, d_j).

In Definition 1, each (m_i, d_i) is called a *resource block*. Condition 2 of Definition 1 specifies that resource blocks in a resource distribution are ordered by the time when a resource block is provided (see Fig. 3 for an illustration). The resource distribution describes how many cores are allocated to the task in different times during the execution. In one resource block, the allocated number of cores remains the same. In different resource blocks, the allocated number of cores may be different. The resource distribution is to model the execution behavior that there may be different numbers of parallel vertices to execute in different times during the execution. After the computing resources are specified by a resource distribution, the parallel real-time task is scheduled by a work-conserving scheduler on the resource distribution.

5.3 Schedulability test for hard real-time tasks

Definition 2 (Critical Path (He et al. 2019)) The critical path $\lambda^* = (\pi_0, \cdots, \pi_k)$ of an execution sequence is a complete path satisfying the following property.

$$\forall \pi_i \in \lambda^* \setminus \{\pi_0\} : f(\pi_{i-1}) = \max_{u \in pred(\pi_i)} \{f(u)\} \tag{8}$$

The critical path is specific to an execution sequence of the parallel task G. A critical path of G in an execution sequence is not necessarily the longest path of G.

Example 4 For the execution sequence in Fig. 1b, a critical path of G is (v_0, v_1, v_4, v_5). In Fig. 1c, a critical path of G is (v_0, v_2, v_4, v_5), which is not the longest path of G.

Lemma 1 *In an execution sequence under work-conserving scheduling, when the critical path is not executing, all cores are busy.*

Proof Suppose that the critical path of this execution sequence is $\lambda^* = (\pi_0, \cdots, \pi_k)$. $\forall i \in (0, k]$: by Definition 2, π_{i-1} is with the maximum finish time among all the predecessors of π_i. This means that when π_{i-1} completes its execution, all predecessors of π_i have completed execution. Therefore, π_i is eligible at $f(\pi_{i-1})$. If some core is idle in $[f(\pi_{i-1}), s(\pi_i)]$, it contradicts the fact that the scheduling is work-conserving. □

Given a parallel real-time task (G, D, T), and a resource distribution $\Theta = \{(m_0, d_0), \cdots, (m_k, d_k)\}$ satisfying $len(G) < \sum_{i \in [0,k]} d_i \leq D$, next we derive a

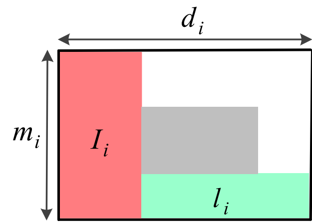


Fig. 4 The notations used in the proof of Theorem 1. Green area represents the critical path. Red area represents the workload executed when the critical path is not executing. Gray area represents other possible workload executed in this resource block

sufficient schedulability test such that task G is schedulable on resource distribution Θ.

We reorder the resource blocks in Θ according to their m_i. Specifically, $\Theta = \{(m_{b_0}, d_{b_0}), \cdots, (m_{b_k}, d_{b_k})\}$ satisfies

$$\forall i, j \in [0, k] \wedge i < j : m_{b_i} \geq m_{b_j} \tag{9}$$

Using Eq. (10), an index s can be determined. Specifically, we can check every value in $[0, k]$ to find the s satisfying Eq. (10).

$$\sum_{i \in [0,s]} d_{b_i} \leq len(G) < \sum_{i \in [0,s+1]} d_{b_i} \tag{10}$$

If $len(G) < d_{b_0}$, let $s = -1$. We define $Q := \{b_i \mid i \in [0, s]\}$, $q := b_{s+1}$, $r := len(G) - \sum_{i \in [0,s]} d_{b_i}$. In particular, if $s = -1$, then $Q = \emptyset$, $q = b_0$, $r = len(G)$.

Theorem 1 *If Eq. (11) holds, then the parallel real-time task* (G, D, T) *is schedulable on resource distribution* $\Theta = \{(m_0, d_0), \cdots, (m_k, d_k)\}$ *under work-conserving scheduling.*

$$vol(G) - len(G) + \sum_{i \in Q} m_i d_i + m_q r \leq \sum_{i \in [0,k]} m_i d_i \tag{11}$$

Proof We prove it by contradiction and assume that task G is not schedulable on resource distribution Θ. Let ε be an execution sequence of G and assume that ε does not complete its execution. We denote the volume of the workload of ε executed on Θ as W_0. Since ε does not complete its execution, we have

$$W_0 < vol(G) \tag{12}$$

Let λ^* be the critical path of ε. We denote the length of λ^* located in (m_i, d_i) as l_i, and denote the volume of workload executed in (m_i, d_i) when λ^* is not executing as I_i (see Fig. 4 for an illustration). With these notations, we have

$$len(\lambda^*) = \sum_{i \in [0,k]} l_i \tag{13}$$

Therefore, the volume of the workload of ε executed on Θ is at least W_1.

$$W_1 := \sum_{i\in[0,k]} (I_i + l_i)$$

Since W_1 is the minimum volume of the workload of ε executed on Θ, we have

$$W_1 \le W_0 \tag{14}$$

By Lemma 1, when λ^* is not executing, all cores are busy. We have $\forall i \in [0,k]$: $I_i = m_i(d_i - l_i)$. Therefore,

$$W_1 = \sum_{i\in[0,k]} (m_i(d_i - l_i) + l_i) = \sum_{i\in[0,k]} m_i d_i - \sum_{i\in[0,k]} (m_i - 1) l_i \tag{15}$$

Similar to Eq. (10), an index s^* can be determined by

$$\sum_{i\in[0,s^*]} d_{b_i} \le len(\lambda^*) < \sum_{i\in[0,s^*+1]} d_{b_i}$$

We also define $Q^* := \{b_i \mid i \in [0, s^*]\}$, $q^* := b_{s^*+1}$, $r^* := len(\lambda^*) - \sum_{i\in[0,s^*]} d_{b_i}$.
By Eqs. (9) and (13), we have

$$\sum_{i\in[0,k]} (m_i - 1) l_i \le \sum_{i\in Q^*} (m_i - 1) d_i + (m_{q^*} - 1) r^* \tag{16}$$

Since $len(\lambda^*) \le len(G)$, we have

$$\sum_{i\in Q^*} (m_i - 1) d_i + (m_{q^*} - 1) r^* \le \sum_{i\in Q} (m_i - 1) d_i + (m_q - 1) r$$

Together with Eqs. (15) and (16), we have

$$W_1 \ge \sum_{i\in[0,k]} m_i d_i - \sum_{i\in Q} (m_i - 1) d_i - (m_q - 1) r$$

Combining with Eqs. (12) and (14), we have

$$\sum_{i\in[0,k]} m_i d_i - \sum_{i\in Q} (m_i - 1) d_i - (m_q - 1) r < vol(G)$$

By Eq. (10), we have

$$\sum_{i\in[0,k]} m_i d_i - \sum_{i\in Q} m_i d_i - m_q r + len(G) < vol(G)$$

Therefore,

$$vol(G) - len(G) + \sum_{i\in Q} m_i d_i + m_q r > \sum_{i\in[0,k]} m_i d_i$$

This contradicts Eq. (11), which completes the proof. □

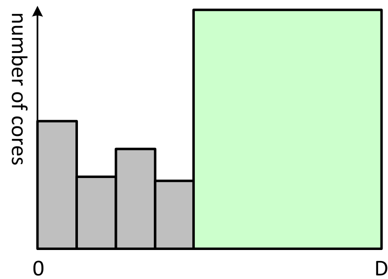


Fig. 5 Illustration of the profiling method in Sect. 5.4

Theorem 1 has straightforward meanings: the RHS (right-hand side) of Eq. (11) is the capacity that the resource distribution can provide; the LHS (left-hand side) of Eq. (11) is the resource requirement such that the parallel real-time task can be schedulable. Theorem 1 can gracefully degrade to the schedulability test in (Baruah 2015d) when there is only one resource block in the resource distribution as shown in Corollary 1.

Corollary 1 *If Eq. (17) holds, then the parallel real-time task (G, D, T) is schedulable on a computing platform with m cores under work-conserving scheduling.*

$$len(G) + \frac{vol(G) - len(G)}{m} \leq D \tag{17}$$

Proof Let the resource distribution Θ in Theorem 1 be $\{(m_0 = m, d_0 = D)\}$. We have that the s in Eq. (10) is -1, and $Q = \emptyset$, $q = b_0 = 0$, $r = len(G)$. Therefore, $\sum_{i \in Q} m_i d_i = 0$ and $m_q = m_0 = m$. Equation (11) degrades to

$$vol(G) - len(G) + mlen(G) \leq mD$$

which is equivalent to Eq. (17). □

5.4 Design method for soft real-time tasks

Under the conditions derived in Sect. 5.3 which ensure that the resource distribution can guarantee the deadlines for hard real-time tasks, this subsection discusses the design method of the resource distribution such that the reclaimed computing resources for soft real-time tasks can be as many as possible.

As stated before, federated scheduling allocates a rectangular piece of computing resources based on the worst-case characterizations of the parallel real-time task. However, during executions, the actually used computing resources may only be a small portion of the allocated computing resources. In this method, we profile the “shape” of executions of the hard real-time task for allocating computing resources to avoid resource-wasting in the typical case (the gray area in Fig. 5) and employ Theorem 1 for allocating an extra piece of computing resources (the green area in Fig. 5) to ensure that the deadline is satisfied in the worst case. In

the following, given a hard real-time task (G, D, T), we present the procedure to determine the resource distribution.

Step 1: decide profiling setting. Let m denote the number of cores on which the task is executed during the profiling. m is computed by Eq. (2). Assume that the task starts execution at time 0. The time interval $[0, D - len(G)]$ is divided into n resource blocks uniformly. For each resource block (m_i, d_i),

$$d_i = \frac{D - len(G)}{n}$$

Step 2: profile the task. The task is executed on m cores during time interval $[0, D - len(G)]$ for many times under a work-conserving scheduler. Note that we execute the task until time point $D - len(G)$ and we do not require the task to complete its execution during the profiling. During the profiling, the following information is recorded.

- The actually used numbers of cores during executions.
- The probability p_i that the task can finish its execution before or within resource block (m_i, d_i).

Suppose that the task is executed for β times. During β times, there are α times that the task finishes its execution before or within resource block (m_i, d_i). Then the probability $p_i = \alpha/\beta$. During one execution, in resource block (m_i, d_i), suppose that the actually used numbers of cores are $\{(y_0, x_0), \cdots, (y_j, x_j), \cdots, (y_k, x_k)\}$, where (y_j, x_j) means that in a time interval of length x_j, the actually used number of cores is y_j. For this execution and resource block (m_i, d_i), the average number of cores is computed as

$$m_i = \frac{\sum_{j \in [0,k]} y_j x_j}{\sum_{j \in [0,k]} x_j} = \frac{\sum_{j \in [0,k]} y_j x_j}{d_i}$$

During all the executions, we compute the average m_i among all executions, and round it to the nearest integer as the final m_i for resource block (m_i, d_i).

Step 3: compute resource distribution. After Step 2, in time interval $[0, D - len(G)]$, we have the profiled resource blocks $\{(m_0, d_0), \cdots, (m_{n-1}, d_{n-1})\}$. Then, Algorithm 1 is employed to compute the final resource distribution, which is $\Theta = \{(m_0, d_0), \cdots, (m_{i^*}, d_{i^*}), (m_{i^*+1} = m(i^*), d_{i^*+1} = d(i^*))\}$. See explanations of the algorithm in the proof of Theorem 2. In Line 3, m is computed in Step 1 using Eq. (2). Line 5 utilizes the probability p_i in Step 2 to compute the average computing resources $A(i)$, which is used as criteria for selecting the resource distributions computed during all the iterations of Algorithm 1. The selection heuristic $A(i)$ is from (Baruah 2018). Note that in Line 1, the loop is for $i \in [0, n-2]$, not including $n - 1$. This is because the total length of all resource blocks is $D - len(G)$, i.e., $\sum_{j \in [0,n-1]} d_j = D - len(G)$. In the loop of Line 1, if we let $i = n - 1$, then in Line 2, the denominator of the formula will be 0. So in the loop of Line 1, we enforce $i \neq n - 1$.

Algorithm 1 Compute resource distribution

1: **for** $i \leftarrow 0, \cdots, n-2$ **do**
2: $\quad m(i) \leftarrow \left\lceil \frac{vol(G)-len(G)-\sum_{j\in[0,i]} m_j d_j}{D-len(G)-\sum_{j\in[0,i]} d_j} \right\rceil$
3: $\quad m(i) \leftarrow \max\{m(i), m\}$
4: $\quad d(i) \leftarrow D - \sum_{j\in[0,i]} d_j$
5: $\quad A(i) \leftarrow \sum_{j\in[0,i]} m_j d_j + (1-p_i)m(i)d(i)$
6: **end for**
7: $i^* \leftarrow$ the i with smallest $A(i)$

Theorem 2 *The resource distribution* $\Theta = \{(m_0, d_0), \cdots, (m_{i^*}, d_{i^*}), (m(i^*), d(i^*))\}$ *computed in Step* 3 *is correct in the sense that the hard real-time task* (G, D, T) *can always meet its deadline under work-conserving scheduling on* Θ.

Proof In Algorithm 1, after each iteration, there is a resource distribution $\{(m_0, d_0), \cdots, (m_i, d_i), (m_{i+1} = m(i), d_{i+1} = d(i))\}$. The resource distribution Θ in Step 3 is one of them. Therefore, to prove Theorem 2, it is sufficient to show that the resource distribution in each iteration of Algorithm 1 is correct. We prove this using Theorem 1.

Since in Step 1, the task is profiled on m cores (m is computed by Eq. (2)), we have $\forall j \in [0, i] : m_j \le m$. By Line 3 of Algorithm 1, we have $m(i) \ge m$. Therefore,

$$\forall j \in [0, i] : m(i) \ge m_j \tag{18}$$

By Step 1, the task is profiled in time interval $[0, D - len(G)]$. Therefore, in Line 4 of Algorithm 1, $\sum_{j\in[0,i]} d_j < D - len(G)$, and we have

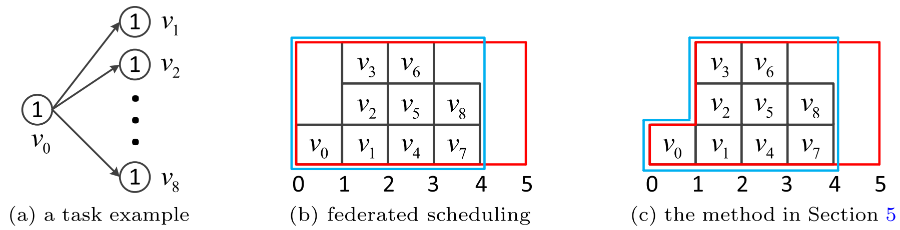
$$d(i) > len(G) \tag{19}$$

By Eqs. (18) and (19), we have that for Theorem 1, in Eq. (10), the $s = -1$. So, $Q = \emptyset$, $q = i + 1$, $r = len(G)$. In this case, by Eq. (11), we have

$$vol(G) - len(G) + m(i)len(G) \le m(i)d(i) + \sum_{j\in[0,i]} m_j d_j$$

which is Line 2 of Algorithm 1. The theorem is proved. □

Note that during each iteration of Algorithm 1, a resource distribution is computed and each resource distribution is with an $A(i)$. So, after Algorithm 1, a list of resource distributions are computed. It should be emphasized that each of these computed resource distributions can guarantee the hard real-time requirement. This is because the equation in Line 2 of Algorithm 1 is derived according to Theorem 1 (see the rigorous reasoning in Theorem 2). The $A(i)$, which is computed in Line 5, only serves as a heuristic to select a "better" resource distribution among the resource distributions computed during all the iterations of Algorithm 1. So we do not need $A(i)$ in the proof of Theorem 2, which is about hard real-time guarantees.



(a) a task example (b) federated scheduling (c) the method in Section 5

Fig. 6 Comparison of the computing resources between federated scheduling and the method in Sect. 5

For the profiling procedure, in Step 2, the task is profiled in time interval $[0, D - len(G)]$, not in [0, D]. The reasons for this choice are as follows. On one hand, the profiling is to capture the typical execution of the task so that we can reclaim computing resources. Since the execution times of vertices may be far less than the worst-case execution time, it is likely that the typical execution will be in the beginning part of executions. In time interval $[D - len(G), D]$, chances are that during the profiling, the actually used number of cores is always zero. On the other hand, the profiling should not jeopardize the hard real-time guarantees. With resource blocks in $[D - len(G), D]$, a valid $m(i)$ in Line 2 of Algorithm 1 cannot be computed (the denominator of the equation in Line 2 will be negative). And recall that the equation in Line 2 is critical for hard real-time guarantees.

In Step 1, the time interval $[0, D - len(G)]$ is equally divided into n resource blocks. For a larger n, the profiled "shape" of executions will be more accurate, which means potentially more reclaimed computing resources. However, since the execution time of vertices may change, the shapes of different executions may be different. Therefore, a larger n also means more migration overhead. What's more, to capture the typical execution of the task, equally dividing the time interval $[0, D - len(G)]$ does not sound to be a wise choice. Exploring the effects of different n and other possible profiling methods is left to future work. In the evaluation, we simply let $n = 3, 4, 5$ to showcase the effectiveness of the proposed method.

5.5 An example

This subsection uses the example in Fig. 6 to explain the concept of resource distribution and the proposed technique in Sect. 5. Figure 6a shows a parallel real-time task G, and let the deadline $D = 5$. In G, the source vertex v_0 spawns eight parallel vertices $v_1, \cdots, v_8$. So the length $len(G) = 2$, and the volume $vol(G) = 9$.

In federated scheduling, by Eq. (2), the allocated number of cores $m = \lceil (9-2)/(5-2) \rceil = 3$. Figure 6b shows a possible execution sequence of G under federated scheduling. For simplicity, we suppose that each vertex executes for its WCET. The allocated computing resources to guarantee its deadline are $m \times D = 3 \times 5 = 15$ (the area covered by red lines). Suppose that after the task completes its execution, the remaining computing resources can be used for soft real-time tasks or the hardware can be placed in low-energy mode to save energy. So the

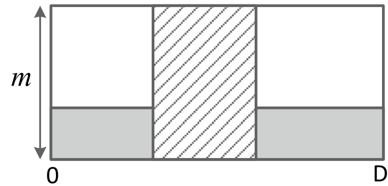


Fig. 7 The computation of the number of cores in federated scheduling

actual computing resources during the execution are $3 \times 4 = 12$ (the area covered by blue lines).

In the proposed technique shown in Fig. 6c, we first profile the task using the procedure in Sect. 5.4. In Step 1, the task will be profiled on $m = 3$ cores during time interval [0, 3]. And we divide this time interval into $n = 3$ blocks. In Step 2, suppose that the profiled resource blocks are $\{(m_0 = 1, d_0 = 1), (m_1 = 3, d_1 = 1), (m_2 = 3, d_2 = 1)\}$. In this simple example, we ignore the probability p_i, and in Algorithm 1 of Step 3, suppose $i^* = 1$. In Line 2 of Algorithm 1, $m(i^*) = \lceil (9 - 2 - 4)/(5 - 2 - 2) \rceil = 3$. In Line 4, $d(i^*) = 5 - 2 = 3$. Therefore, the final resource distribution is $\{(m_0 = 1, d_0 = 1), (m_1 = 3, d_1 = 1), (m_2 = m(i^*) = 3, d_2 = d(i^*) = 3)\}$. In our method, the allocated computing resources to guarantee its deadline are $1 \times 1 + 3 \times 1 + 3 \times 3 = 13$ (the area covered by red lines in Fig. 6c). The actual computing resources during the execution are $1 \times 1 + 3 \times 1 + 3 \times 2 = 10$ (the area covered by blue lines).

In this example, it can be seen that compared to federated scheduling, the proposed technique reduces both the *allocated computing resources* (i.e., the computing resources determined before execution to guarantee the deadline) and the *actual computing resources* (i.e., the computing resources determined using the runtime information).

6 Reclaiming resources in the ending

In this technique, during the execution of the parallel real-time task, we collect information regarding the execution of the hard real-time task and gradually reduce the allocated number of cores to reclaim computing resources in the ending part of executions. The time points to reduce the number of cores during executions are characterized by a concept called *allocation vector*.

6.1 Intuition of the technique

As stated in Sect. 4, federated scheduling uses the bound in Eq. (1) to allocate the number of cores. It enforces the bound in Eq. (1) to be less than or equal to the deadline D to compute the number of cores m. This computation of the number of cores is achieved by assuming two types of execution: (1) only the longest path is executing (the gray solid area in Fig. 7); (2) other workload is executing and occupies all of the cores (the dashed area in Fig. 7). However, during real executions, the longest

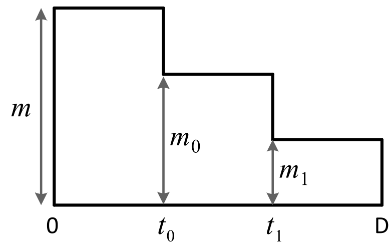


Fig. 8 Illustration of the scheduling for the technique in Sect. 6

path and other workload may execute in parallel, which means that there is another type of execution, i.e., more than one core are busy but not all cores are busy. It is possible to monitor this type of execution during runtime and utilize the monitored information to reduce the allocated number of cores dynamically (see Fig. 8), thus reclaiming computing resources for soft real-time tasks or saving energy by placing the hardware in low-energy mode.

What is more, in this ladder-like resource allocation, the number of cores is reduced before the task completes its execution. This means that the computing resources are reclaimed earlier than the finish time of the task. The earlier reclaimed computing resources can make more soft real-time tasks finish before their soft deadlines.

Deciding the allocation vector lies in the center of the proposed technique in this section. As stated before, the purposes of this paper are to, first guarantee the deadline for hard real-time tasks, and second reclaim computing resources for soft real-time tasks. In the following, Sect. 6.2 formally defines the allocation vector. Section 6.3 derives the conditions that the adjusted number of cores should satisfy during the allocation vector to guarantee the deadlines for hard real-time tasks. Section 6.4 discusses the heuristics for designing the allocation vector such that the reclaimed computing resources for soft real-time tasks can be as many as possible. Section 6.5 provides an example to explain the proposed technique.

6.2 The scheduling algorithm

Definition 3 (Allocation Vector) For a parallel real-time task (G, D, T), the allocation vector Φ is a set of time points $\{t_0, \cdots, t_k\}$ $(k \geq 0)$ satisfying all of the following conditions.

1. $\forall i \in [0, k]$: $0 \leq t_i < D$.
2. $\forall i, j \in [0, k]$ and $i < j$: $t_i < t_j$.

In an allocation vector, each t_i is called an *allocation point*. The allocation vector is a list of time points. At each time point, the scheduler may adjust the allocated number of cores based on the monitored execution information to release cores earlier. Given a parallel real-time task (G, D, T) and the allocation vector

$\Phi = \{t_0, \cdots, t_k\}$, the scheduling starts at time 0 with the number of cores m computed by Eq. (2). During the scheduling, two types of information are monitored.

1. $w(t)$: the volume of the workload executed from time point 0 to time point t.
2. $l(t)$: the cumulative length of time intervals during which at least one core is idle from time point 0 to time point t.

At each allocation point t_i, if G does not complete its execution, we adjust the allocated number of cores to m_i based on the information monitored above. The method for computing m_i will be detailed in Sect. 6.3. The parallel real-time task is scheduled by a work-conserving scheduler as illustrated in Fig. 8.

6.3 Schedulability test for hard real-time tasks

Lemma 2 *In an execution sequence under work-conserving scheduling, when at least one core is idle, the critical path is executing.*

Proof Lemma 2 is the contrapositive of Lemma 1. □

Theorem 3 *At each allocation point t_i ($i \in [0, k]$), if the allocated number of cores m_i is computed by Eqs. (20) and (21), then the parallel real-time task G is schedulable under work-conserving scheduling with the allocation vector $\Phi = \{t_0, \cdots, t_k\}$.*

if $vol(G) - w(t_i) \leq len(G) - l(t_i)$, *then*

$$m_i = 1 \tag{20}$$

else

$$m_i = \left\lceil \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{D - t_i - len(G) + l(t_i)} \right\rceil \tag{21}$$

Proof Let ε be the execution sequence under analysis of G. At allocation point t_i, we focus on the remaining graph G_i of G (i.e, the part of G that has not been executed until t_i). We denote the critical path of ε as λ^*. Since l_i is the cumulative length of time intervals before t_i where at least one core is idle, by Lemma 2, λ^* is executing in these time intervals. Therefore, the length of λ^* in G_i (i.e., the length of λ^* executing after t_i) is at least $len(\lambda^*) - l(t_i)$, which is bounded by

$$len(G) - l(t_i)$$

And the volume of G_i is bounded by

$$vol(G) - w(t_i)$$

By Eq. (1), the response time of G_i is bounded by

$$len(G) - l(t_i) + \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{m_i}$$

The new deadline of G_i is $D - t_i$. Let

$$len(G) - l(t_i) + \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{m_i} \leq D - t_i$$

which means Eq. (21). □

Note that the volume $vol(G)$ and length $len(G)$ in Theorem 3 are determined offline. We do not need to monitor these two parameters online.

Corollary 2 *The schedulability test in Theorem 3 dominates the test in (Li et al. 2014) (shown in Eqs. (1) and (2)) in the sense that the computing resource allocated by Theorem 3 is no larger than that of (Li et al. 2014).*

Proof It is sufficient to show that $\forall i \in [0, k]$: the m_i in Eq. (21) is no larger than the m in Eq. (2), i.e.,

$$\left\lceil \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{D - t_i - len(G) + l(t_i)} \right\rceil \leq \left\lceil \frac{vol(G) - len(G)}{D - len(G)} \right\rceil \tag{22}$$

Suppose $t_0 = 0$, we have $w(0) = 0$, $l(0) = 0$, so Eq. (22) holds trivially. Therefore, to prove Eq. (22), it suffices to show that $\forall i \in [0, k)$: $m_{i+1} \leq m_i$, i.e.,

$$\frac{vol(G) - w(t_{i+1}) - len(G) + l(t_{i+1})}{D - t_{i+1} - len(G) + l(t_{i+1})} \leq \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{D - t_i - len(G) + l(t_i)} \tag{23}$$

We define

$$\Delta(t) := t_{i+1} - t_i$$
$$\Delta(w) := w(t_{i+1}) - w(t_i)$$
$$\Delta(l) := l(t_{i+1}) - l(t_i)$$

Equation (23) can be rewritten as Eq. (24).

$$\frac{vol(G) - w(t_i) - len(G) + l(t_i) - (\Delta(w) - \Delta(l))}{D - t_i - len(G) + l(t_i) - (\Delta(t) - \Delta(l))} \leq \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{D - t_i - len(G) + l(t_i)} \tag{24}$$

Be aware of the following statement: for $0 < x_1 < x_2, 0 < y_1 < y_2$,

$$\frac{x_1}{y_1} \geq \frac{x_2}{y_2} \Rightarrow \frac{x_2 - x_1}{y_2 - y_1} \leq \frac{x_2}{y_2}$$

Therefore, to prove Eq. (24), it suffices to show that

$$\frac{\Delta(w) - \Delta(l)}{\Delta(t) - \Delta(l)} \geq \frac{vol(G) - w(t_i) - len(G) + l(t_i)}{D - t_i - len(G) + l(t_i)} \tag{25}$$

The length of time interval $[t_i, t_{i+1}]$ is $\Delta(t)$. During time interval $[t_i, t_{i+1}]$, the allocated number of cores is m_i. The volume of the workload executed in $[t_i, t_{i+1}]$ is $\Delta(w)$. By the definitions of $l(t)$ and $\Delta(l)$, $\Delta(l)$ is the cumulative length of time intervals during which at least one core is idle in $[t_i, t_{i+1}]$. Therefore, we have

$$\Delta(w) \geq m_i(\Delta(t) - \Delta(l)) + \Delta(l) \tag{26}$$

which means

$$\frac{\Delta(w) - \Delta(l)}{\Delta(t) - \Delta(l)} \geq m_i$$

Therefore, Eq. (25) holds, which means that Eq. (22) holds. □

6.4 Design method for soft real-time tasks

When encountering an allocation point in the allocation vector, Sect. 6.3 presents how to adjust the allocated number of cores such that hard real-time guarantees are met. This subsection discusses how to design the allocation vector to effectively reclaim computing resources for soft real-time tasks. So the objective is that given a hard real-time task, at each allocation point t_{i+1}, we want to reduce the number of cores m_{i+1} compared to m_i. By Corollary 2, we know that this reduction of the number of cores lies in Eq. (26). Also, as explained in Sect. 6.1, the type of execution that federated scheduling does not assume is critical for reducing the number of cores. Therefore, during the execution, the type of execution satisfying both of the following conditions can help decide whether an allocation point should be placed or not.

1. During the execution, more than one core is busy.
2. During the execution, at least one core is idle.

The more executions belonging to this type, the more the number of cores can be reduced. With this guideline, the allocation vector can be statically determined offline or dynamically determined during runtime. A carefully chosen allocation vector may reduce the number of cores more effectively and incur less vertex migration overhead. In this paper, for the simplicity of the monitoring procedure and the scheduler, we do not monitor the above type of execution and simply check whether the number of cores can be reduced whenever a vertex of the parallel real-time task completes its execution.

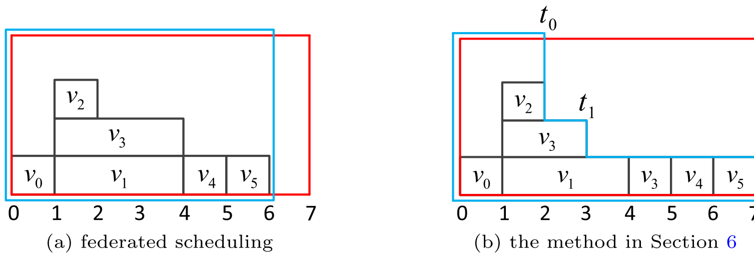


Fig. 9 Comparison of the computing resources between federated scheduling and the method in Sect. 6

6.5 An example

This subsection provides an example to illustrate the usefulness of the allocation vector interface and the effectiveness of the proposed technique in Sect. 6. For the parallel real-time task G in Fig. 1a, suppose that the deadline $D = 7$. The volume $vol(G) = 10$ and the length $len(G) = 6$. Figure 9 shows the possible execution sequences and the computing resources under federated scheduling and the proposed technique in this section.

In Fig. 9a, by Eq. (2), the allocated number of cores $m = \lceil (10 - 6)/(7 - 6) \rceil = 4$. So the allocated computing resources to ensure the deadline are $m \times D = 4 \times 7 = 28$ (the area covered by red lines). Same as Sect. 5.5, we also suppose that after the task completes its execution, the remaining computing resources can be used for soft real-time tasks or the hardware can be placed in low-energy mode to save energy. The actual computing resources during the execution are $4 \times 6 = 24$ (the area covered by blue lines).

In Fig. 9b, suppose that the allocation vector is $\Phi = \{t_0 = 2, t_1 = 3\}$. Here, we statically determine the allocation vector before execution to better explain the scheduling, not like Sect. 6.4, where the allocation vector is determined dynamically during runtime. At time point 0, same as federated scheduling, the allocated number of cores $m = 4$. At allocation point $t_0 = 2$, the monitored information is $w(t_0) = 4$, $l(t_0) = 2$. $vol(G) - w(t_0) = 6$, $len(G) - l(t_0) = 4$ and $D - t_0 = 5$. By Eq. (21), the number of cores is adjusted to $m_0 = \lceil (6 - 4)/(5 - 4) \rceil = 2$. At allocation point $t_1 = 3$, the monitored information is $w(t_1) = 6$, $l(t_1) = 2$. $vol(G) - w(t_1) = 4$, $len(G) - l(t_1) = 4$ and $D - t_1 = 4$. Since $vol(G) - w(t_1) \leq len(G) - l(t_1)$, by Eq. (20), the number of cores is adjusted to $m_1 = 1$. Note that in this technique, the allocated number of cores determined before execution is the same as federated scheduling. The adjusted numbers of cores (i.e., m_0, m_1) are computed based on the information collected during runtime. So the allocated computing resources are $m \times D = 4 \times 7 = 28$ (the area covered by red lines), which is the same as federated scheduling. The actual computing resources are $4 \times 2 + 2 \times 1 + 1 \times 4 = 14$ (the area covered by blue lines). In this example, compared to federated scheduling, the technique in Sect. 6 reclaims $(24 - 14)/24 = 41.7\%$ computing resources for soft real-time tasks (or for placing the hardware in low-energy mode to save energy) and guarantees that the hard real-time task meets its deadline.

7 The proposed approach

The proposed approach employs the two techniques introduced in Sects. 5 and 6 to reclaim computing resources. Jointly employing these two techniques lies in how to determine the two interfaces, i.e, the resource distribution and the allocation vector. The proposed approach is detailed as follows; the scheduling algorithm is always work-conserving.

1. Before the execution, utilize the procedure in Sect. 5.4 to determine the resource distribution $\Theta = \{(m_0, d_0), \cdots, (m_{i^*}, d_{i^*}), (m_{i^*+1} = m(i^*), d_{i^*+1} = d(i^*))\}$.
2. During the execution, all allocation points in the allocation vector are in the last resource block, i.e., resource block (m_{i^*+1}, d_{i^*+1}). For the execution in resource blocks $(m_0, d_0), \cdots, (m_{i^*}, d_{i^*})$, the information specified in Sect. 6.2 is monitored. For the execution in resource block (m_{i^*+1}, d_{i^*+1}), there is an allocation point whenever a vertex of the parallel real-time task completes its execution.

During the execution, at each allocation point t_{j+1}, the computations that should be conducted are: 1) use Eq. (21) to compute the new number of cores m_{j+1}; 2) if $m_{j+1} < m_j$, reduce the number of cores to m_{j+1}. We argue that the overhead of the first part computation (i.e., compute a new m_{j+1}) is negligible compared to the overhead of scheduling decisions, since computing such an equation as Eq. (21) should be quite fast. And the second part computation (i.e., reduce the number of cores to m_{j+1}) is conducted for no more than the number of cores, because every time the second part computation is conducted, the number of cores is reduced by at least one core.

Concerning the hard real-time guarantees, the correctness of the proposed approach follows directly from Theorems 1 and 3.

The primary cause of the four types of pessimism summarized in Sect. 4 is that existing approaches rely on static information of the parallel real-time task (i.e., the graph structure or the worst-case execution time) to allocate computing resources. So the offline analysis has to reserve overly-pessimistic computing resources for the worst-case execution scenarios to ensure hard real-time guarantees. In contrast, our approach combines offline analysis and online monitoring: first the offline analysis reserves enough computing resources for the worst-case execution scenarios to ensure hard real-time guarantees; second, the online monitoring reclaims computing resources based on the information from the actual execution scenarios. Take Type-4 pessimism as an example. Different execution flows of a conditional DAG task can have quite diversified characterizations, so the offline analysis must reserve computing resources based on an approximation of the worst-case scenarios. However, during actual executions, only one execution flow will exist. The online monitoring procedure will know such information (for example, know that the task completes execution earlier in some resource block so that the following resource blocks can be reclaimed or know the actual $w(t)$, $l(t)$ so that some cores can be released earlier) and utilize it to reclaim computing resources for soft real-time tasks. In summary, by combining offline analysis and online monitoring, static information and

runtime information are jointly utilized in our approach to address the four types of pessimism in Sect. 4.

7.1 Overhead considerations

During runtime, there are two types of overheads in the proposed approach: migration overhead and monitoring overhead.

Both techniques in Sects. 5 and 6 can incur migration overhead. For the technique in Sect. 5, different resource blocks may have different numbers of cores. When the number of cores in the next resource block is smaller than the number of cores in the current resource block, vertex migration may happen. In the worst case, the number of migrations is $O(mn)$, where m is the number of cores and n is the number of resource blocks. In our approach, we generally choose $n \leq 5$, which is quite small. And in the end of each resource block, the number of migrations may be far less than m; what's more, migration can happen only when the number of cores decreases. For the technique in Sect. 6, the number of migrations is upper bounded by the number of cores in the last resource block. Note that in our approach, allocation points are the time points when a vertex completes its execution. Therefore, the scheduler only tries to decrease the number of cores when a vertex finishes execution and releases the core that it occupied previously. So in each allocation point, only when the scheduler tries to decrease two or more cores, migration can happen; when the scheduler decreases one core, no migration can happen. Based on the above analysis, we believe that the actual number of migrations during execution is very small. This part of overhead is evaluated in Sect. 8.

The monitoring overhead is related to the technique in Sect. 6. Our approach assumes that $w(t)$ and $l(t)$ (see Sect. 6.2) are monitored so that such information can be utilized for the scheduling decisions. To analyze this part of overhead, the critical issue is how to monitor $w(t)$ and $l(t)$ or what is the time complexity of such monitoring. Many operating systems provide functions to query the busy time of each core, which would be sufficient to calculate $w(t)$ and $l(t)$. Nevertheless, it can be difficult to know the time complexity of such functions in various operating systems. Here, we outline a software approach to monitor $w(t)$ and $l(t)$, and analyze its complexity. Since the scheduler puts a vertex to execute on a certain core and a vertex will notify the scheduler when it finishes execution, the scheduler knows which vertex executes on which cores at every time point. So the scheduler has sufficient information to compute $w(t)$ and $l(t)$. During the scheduling, let *CUR* denote the time point when the current scheduling decision is to be made. The scheduler maintains a variable *TIME* and a list whose length is m (m is the number of cores). Variable *TIME* stores the time point when the latest scheduling decision was made. Each entry of the list is for a core and has one bit of information that stores whether the corresponding core is busy or idle between *TIME* and *CUR*. Note that the state of a core (i.e., busy or idle) remains unchanged between *TIME* and *CUR*. This is because the state of a core can only be changed in time points when a scheduling decision happens and *TIME* is the time point when the latest scheduling decision was made. At each *CUR*, let m' denote the number of cores that are busy between *TIME* and *CUR*. Using the list

maintained by the scheduler, m' can be computed by traversing the list. Let m denote the current total number of cores. We have

$$w(CUR) = w(TIME) + m'(CUR - TIME)$$

And if $m' < m$,

$$l(CUR) = l(TIME) + CUR - TIME$$

else

$$l(CUR) = l(TIME)$$

Finally, after making scheduling decisions, the scheduler updates the list and let $TIME \leftarrow CUR$. Therefore, with such data structures (i.e., $TIME$ and the list), at each CUR, the scheduler can compute $w(t)$ and $l(t)$ incrementally with time complexity of $O(m)$. We believe that the monitoring with time complexity of $O(m)$ and the calculation of Eq. (21) are quite fast and their overhead is negligible compared to the overhead of scheduling decisions and vertex migrations.

8 Evaluation

This section evaluates the proposed approach. The parallel tasks are randomly generated and simulated to observe the reclaimed computing resources. The following approaches are compared.

- OUR. The approach presented in Sect. 7.
- SAN. The approach proposed in (Baruah 2018; Agrawal and Baruah 2018).

As stated in Sect. 2, (Agrawal and Baruah 2018) provides the analysis method to guarantee that the hard real-time task meets its deadline when dynamically adjusting the number of cores. Given a parallel task, the worst-case characterizations (i.e., the volume and length) can be computed. However, since (Agrawal and Baruah 2018) does not specify how to compute the typical-case characterizations, the allocated number of cores cannot be decided in the approach of (Agrawal and Baruah 2018). Given a parallel task, the method of deciding the allocated number of cores is provided in (Baruah 2018) by offline profiling the response time of the task.

Task Generation. The DAG tasks are generated using the Erdös-Rényi method (Cordeiro et al. 2010). First, the number of vertices is determined, which is generated randomly from [20, 100]. Second, a parameter pf called *parallelism factor* is determined, which is generated randomly from [0.1, 0.9]. Third, the edges of the graph are generated as follows: for each pair of vertices, we generate a random value in [0, 1] and add an edge between this pair of vertices if the generated value is less than pf. A larger pf means that there are more edges generated in the graph. So the larger pf, the more sequential the graph is. Fourth, the volume of the task is determined, which is generated randomly from [1000, 3000]. Once the volume

is determined, the UUnifast method (Bini and Buttazzo 2005) is used to distribute the volume to WCETs of vertices. Now the DAG is generated, so the length of the longest path $len(G)$ can be computed. Fifth, the number of cores is determined, which is generated randomly from [2, 8]. Sixth, by (2), the deadline D is computed by $len(G) + \frac{vol(G)-len(G)}{m}$, where m is the number of cores. The default settings are summarized as follows. The volume $vol(G)$, the parallelism factor pf, the number of cores m, and the vertex number $|V|$ are randomly and uniformly drawn in [1000, 3000], [0.1, 0.9], [2, 8] and [20, 100], respectively.

Evaluation Method. The compared approaches are based on profiling or monitoring the executions of parallel tasks. We simulate the execution of the generated parallel tasks for profiling or monitoring. During the simulation, the execution time of vertices is sampled based on its WCET under the Gumbel distribution (Edgar and Burns 2001). The scheduling algorithm is work-conserving. If there are multiple eligible vertices waiting to be executed, we randomly choose one vertex for execution. So during different executions of a task, the scheduling decisions can be different. For each parameter configuration (i.e., each data point in the figures), 1000 simulations are done to compute the average performance for each compared approach. For the profiling procedure in Sect. 5.4, in Step 1, we evaluate $n = 3, 4, 5$; in Step 2, a task is profiled 100 times.

Evaluation Metric. The allocated computing resources and the actual computing resources are used as the metrics to compare the approaches. The allocated computing resources are the computing resources allocated to a parallel task before the executions to ensure its deadline. This metric can be decided only using the static information of the parallel task. The actual computing resources are the computing resources occupied by a parallel task during an execution. The runtime information of an execution is needed to decide this metric. See the examples in Sects. 5.5 and 6.5 for detailed explanations.

Evaluation Result. Figure 10 reports the allocated computing resources of the two approaches. For the y-axis, the allocated computing resources are normalized to the volume of the task with $n = 4$. The results with changing the parallelism factor pf are in Fig. 10a. When pf increases, the task becomes more sequential, which means that the length of the task increases. Recall that in Sect. 5.4, the task is profiled in $[0, D - len(G)]$. Since the duration of the time interval for profiling decreases, the profiling procedure gets less effective. This leads to more allocated computing resources. Therefore, when pf increases, the allocated computing resources of our method increase. In Fig. 10b, more computing resources are allocated as the number of cores increases. This is typical for parallel computing: the more parallel the task is, the more computing resources are wasted. How to properly utilize the computing power of multicores and achieve linear speedup is always a major challenge for parallel computing. Since the offline analysis and online monitoring have nothing to do with the number of vertices, the two approaches are unrelated to the number of vertices in the task. Figure 10c also backs this observation. Figure 10 shows that our approach allocates less computing resources than SAN. This demonstrates that our profiling method in Sect. 5.4 is superior to the profiling method in (Baruah 2018).

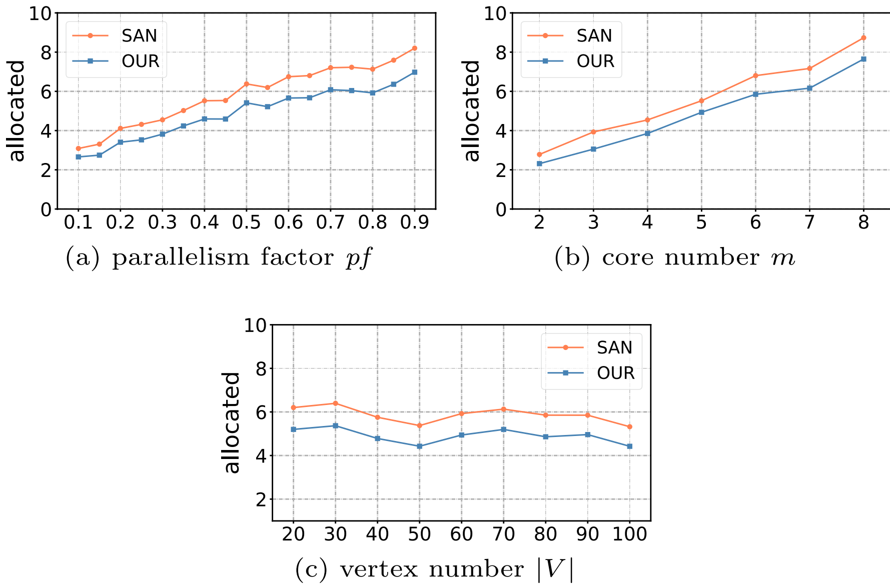


Fig. 10 Evaluation with allocated computing resources (n = 4)

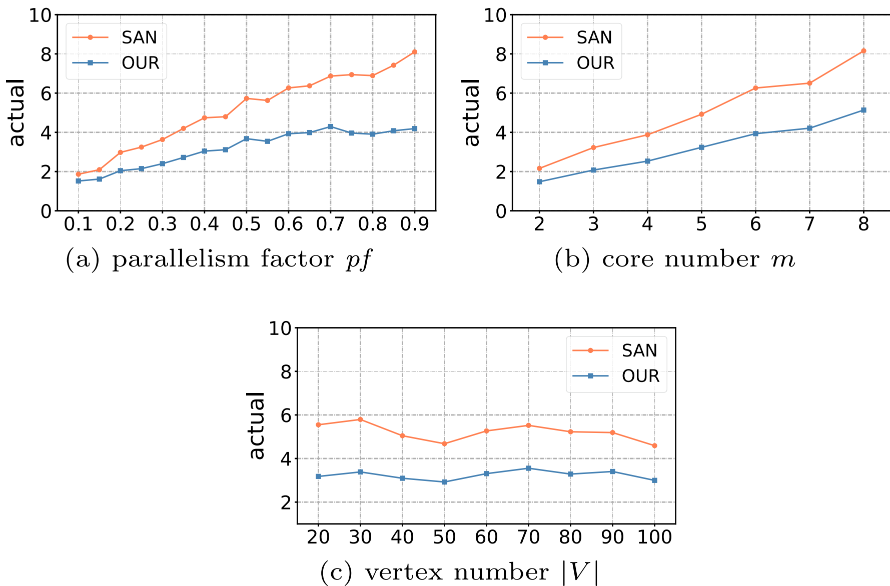


Fig. 11 Evaluation with actual computing resources (n = 4)

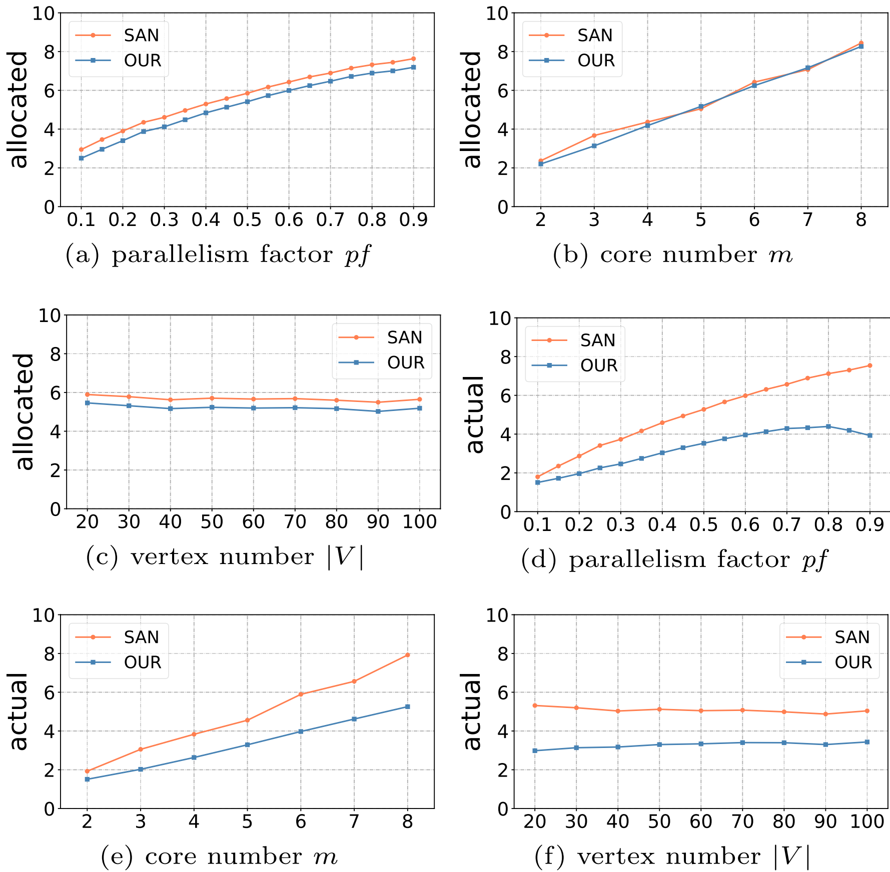


Fig. 12 Evaluation with allocated and actual computing resources (n = 3)

Figure 11 reports the actual computing resources of the two approaches with $n = 4$. For the y-axis, the actual computing resources are normalized to the workload of executions. Since during simulations, the execution time is less than the WCET, the workload of an execution is less than the volume of the task. Reclaiming computing resources during runtime is the main purpose of this paper. The less the actual computing resources, the more the reclaimed computing resources. Figure 11 shows similar trends as the corresponding figures in Fig. 10. And the reasons of the trends in Fig. 11 are the same as the reasons of the corresponding figures in Fig. 10. In Fig. 11a, when pf becomes larger, our approach is still rather effective, reclaiming significantly more resources than SAN. This is because the method in Sect. 6 can release the number of cores timely when these cores are no longer necessary to guarantee the deadline. Compared to SAN, our approach can reduce the actual computing resources by up to 48.3% with $pf = 0.9$. In Fig. 11c, compared to SAN, our approach reduces the actual computing resources by 37.8% on average.

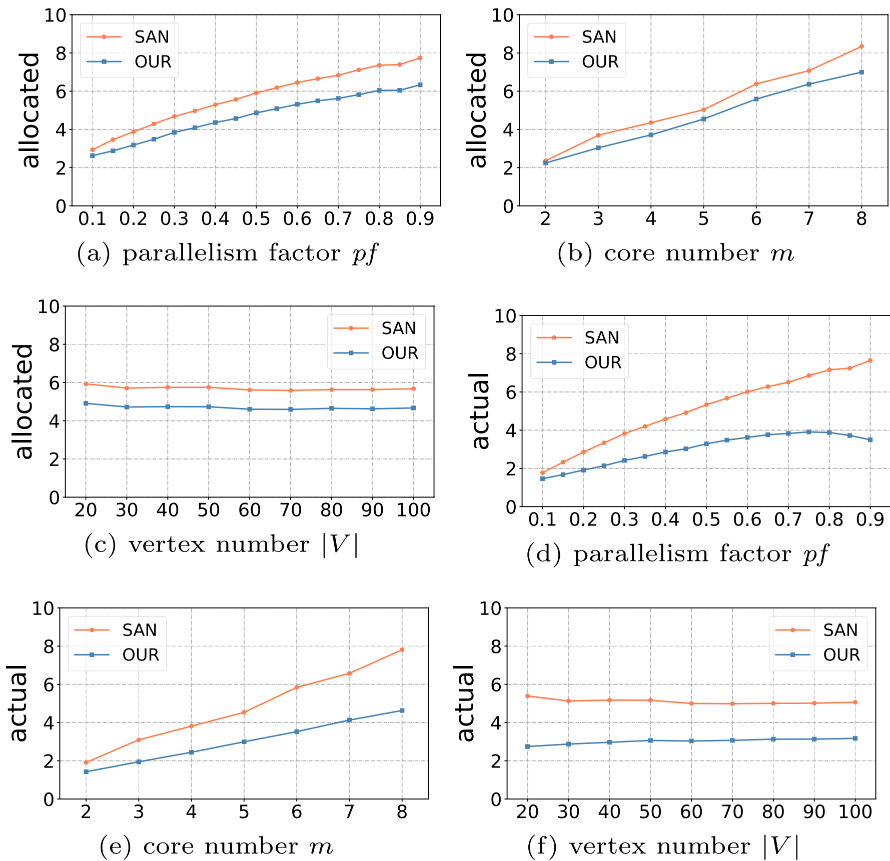


Fig. 13 Evaluation with allocated and actual computing resources (n = 5)

How the different number of resource blocks in the profiling procedure affects the performance is also evaluated and the results are reported in Figs. 12 and 13. The experimental settings are the same as Fig. 10 and Fig. 11 except for the number of resource blocks n. The evaluation shows that regarding the allocated computing resources, compared to SAN, the performance improvement of our approach with $n = 3$ is smaller than $n = 4, 5$. This is because SAN always allocates two resource blocks. So when n is smaller, our approach is closer to SAN regarding the allocated computing resources. We observe that regarding the actual computing resources, the performance improvement of our approach is almost unchanged for different n. This is mainly because the technique in Sect. 6, which is unrelated to n, has a large influence on the actual computing resources.

Migration Overheads. Experiments are also done to see how the overhead of our approach affects the performances. As mentioned in Sect. 7.1, compared to SAN, the proposed approach has additional migration overhead and monitoring overhead. Since the effects of overhead can only be measured during runtime, we compare the

actual computing resources to evaluate the effects of overhead. The experimental settings are the same as Fig. 11. For migration overhead, whenever a vertex migration happens, we add 5% of the WCET of the vertex to the WCET of the vertex for the simulation. As analyzed in Sect. 7.1, the monitoring overhead is with time complexity $O(m)$, where m is the number of cores. In our simulation, $m \leq 8$. So the monitoring overhead should be negligible to the scheduling decisions. What's more, it is difficult to decide how such a small overhead can be compared to the WCET of vertices so that such overhead can be accounted in the simulation as the migration overhead does. Therefore, in the experiment, we only consider migration overhead. The resulting figures are almost identical to Fig. 11 and thus not included in the paper. We observed that the performance decrease of our approach is less than 1%. Consistent with the analysis in Sect. 7.1, this is because the actual number of migrations is very small during simulations, so the performance of our approach is almost unaffected by the overhead.

In summary, by fully exploring the ladder-like resource allocation, with maintaining hard real-time guarantees, our approach can reduce the actual computing resources by 48.3% at the maximum and by 37.8% on average, significantly reclaiming more computing resources than the approach in (Agrawal and Baruah 2018; Baruah 2018).

9 Conclusion

In this paper, to address the resource-wasting problem in federated scheduling, we propose an approach by offline profiling and online monitoring the executions of the hard parallel real-time task and dynamically adjusting the allocated number of cores, thus reclaiming computing resources for soft real-time tasks or placing the reclaimed computing resources to low-energy mode for saving energy. Experiments demonstrate that the proposed approach can reclaim significantly more computing resources than existing approaches while maintaining hard real-time guarantees. This work only considers the scheduling of a single hard real-time task. In future, making use of the ladder-like resource allocation, this work can function as a starting point for other research directions, such as mixed-criticality scheduling (since Sect. 6.1 suggests that the method can release computing resources earlier, thus making more soft real-time tasks finish before their soft deadlines) or more competitive hard real-time scheduling algorithms (since Sect. 5.1 suggests that two parallel tasks may share cores to allow more hard real-time tasks schedulable).

Funding Open access funding provided by The Hong Kong Polytechnic University.

Data availability The data that support the findings of this study are available on request from the corresponding author.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose. The authors have no conflict of interest to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Agrawal K, Baruah S (2018) A measurement-based model for parallel real-time tasks. In: 30th Euromicro conference on real-time systems (ECRTS 2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

Baruah S (2015a) The federated scheduling of constrained-deadline sporadic DAG task systems. In: 2015 design, automation & test in Europe conference & exhibition (DATE), IEEE, pp 1323–1328

Baruah S (2015b) Federated scheduling of sporadic DAG task systems. In: 2015 IEEE international parallel and distributed processing symposium, IEEE, pp 179–186

Baruah S (2015c) The federated scheduling of systems of conditional sporadic DAG tasks. In: Proceedings of the 12th international conference on embedded software, IEEE Press, pp 1–10

Baruah S (2015d) The federated scheduling of systems of conditional sporadic DAG tasks. In: 2015 international conference on embedded software (EMSOFT), IEEE, pp 1–10

Baruah S (2018) Resource-efficient execution of conditional parallel real-time tasks. In: European conference on parallel processing, Springer, pp 218–231

Baruah S, Bonifaci V, Marchetti-Spaccamela A (2015) The global EDF scheduling of systems of conditional sporadic DAG tasks. In: 2015 27th Euromicro conference on real-time systems, IEEE, pp 222–231

Bernat G, Colin A, Petters SM (2002) WCET analysis of probabilistic hard real-time systems. In: 23rd IEEE real-time systems symposium, 2002. RTSS 2002., IEEE, pp 279–288

Bi R, He Q, Sun J, et al (2022) Response time analysis for prioritized DAG task with mutually exclusive vertices. In: 2022 IEEE real-time systems symposium (RTSS), IEEE, pp 460–473

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-time Syst 30(1–2):129–154

Casini D, Biondi A, Nelissen G, et al (2018) Partitioned fixed-priority scheduling of parallel tasks without preemptions. In: 2018 IEEE real-time systems symposium (RTSS), IEEE, pp 421–433

Chen P, Liu W, Jiang X et al (2019) Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems. ACM Trans Embedded Comput Syst (TECS) 18(5s):1–19

Chen P, Liu W, Chen H et al (2020) Reduced worst-case communication latency using single-cycle multihop traversal network-on-chip. IEEE Trans Comput Aided Design Integr Circuits Syst 40(7):1381–1394

Cordeiro D, Mounié G, Perarnau S, et al (2010) Random graph generation for scheduling simulations. In: 3rd international ICST conference on simulation tools and techniques (SIMUTools 2010), ICST, p 10

Dong Z, Liu C (2019) Analysis techniques for supporting hard real-time sporadic gang task systems. Real-Time Syst 55:641–666

Edgar S, Burns A (2001) Statistical analysis of WCET for scheduling. In: Proceedings 22nd IEEE real-time systems symposium (RTSS 2001)(Cat. No. 01PR1420), IEEE, pp 215–224

Fonseca J, Nelissen G, Nelis V, et al (2016) Response time analysis of sporadic DAG tasks under partitioned scheduling. In: 2016 11th IEEE symposium on industrial embedded systems (SIES), IEEE, pp 1–10

Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM J Appl Math 17(2):416–429

Han M, Guan N, Sun J et al (2019) Response time bounds for typed DAG parallel tasks on heterogeneous multi-cores. IEEE Trans Parallel Distrib Syst 30(11):2567–2581

He Q, Jiang X, Guan N et al (2019) Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores. IEEE Trans Parallel Distrib Syst 30(10):2283–2295

He Q, Lv M, Guan N (2021) Response time bounds for DAG tasks with arbitrary intra-task priority assignment. In: 33rd Euromicro conference on real-time systems (ECRTS 2021), Schloss Dagstuhl-Leibniz-Zentrum für Informatik
He Q, Guan N, Lv M, et al (2022) Bounding the response time of DAG tasks using long paths. In: 2022 IEEE real-time systems symposium (RTSS), IEEE, pp 474–486
He Q, Guan N, Lv M, et al (2023a) On the degree of parallelism in real-time scheduling of DAG tasks. In: 2023 design, automation & Test in Europe conference & exhibition (DATE), IEEE, pp 1–6
He Q, Guan N, Lv M et al (2023b) The shape of a DAG: bounding the response time using long paths. Real-Time Syst 2023:1–40
He Q, Sun J, Guan N et al (2023c) Real-time scheduling of conditional DAG tasks with intra-task priority assignment. IEEE Trans Comput Aided Design Integr Circuits Syst 42(10):3196–3209. https://doi.org/10.1109/TCAD.2023.3241221
Jiang X, Long X, Guan N, et al (2016) On the decomposition-based global EDF scheduling of parallel real-time tasks. In: 2016 IEEE real-time systems symposium (RTSS), IEEE, pp 237–246
Jiang X, Guan N, Long X, et al (2017) Semi-federated scheduling of parallel real-time tasks on multiprocessors. In: 2017 IEEE real-time systems symposium (RTSS), IEEE, pp 80–91
Jiang X, Guan N, Long X et al (2020) Real-time scheduling of parallel tasks with tight deadlines. J Syst Archit 108:101742
Jiang X, Guan N, Liang H, et al (2021) Virtually-federated scheduling of parallel real-time tasks. In: 2021 IEEE real-time systems symposium (RTSS), IEEE, pp 482–494
Lakshmanan K, Kato S, Rajkumar R (2010) Scheduling parallel real-time tasks on multi-core processors. In: 2010 31st IEEE real-time systems symposium, IEEE, pp 259–268
Lee S, Lee S, Lee J (2022) Response time analysis for real-time global gang scheduling. In: 2022 IEEE real-time systems symposium (RTSS), IEEE, pp 92–104
Li J, Agrawal K, Lu C, et al (2013) Analysis of global EDF for parallel tasks. In: 2013 25th Euromicro conference on real-time systems, IEEE, pp 3–13
Li J, Chen JJ, Agrawal K, et al (2014) Analysis of federated and global scheduling for parallel real-time tasks. In: 2014 26th Euromicro conference on real-time systems, IEEE, pp 85–96
Li R, Guan N, Jiang X, et al (2022) Worst-case time disparity analysis of message synchronization in ros. In: 2022 IEEE real-time systems symposium (RTSS), IEEE, pp 40–52
Liang H, Jiang X, Guan N, et al (2023) Response time analysis and optimization of DAG tasks exploiting mutually exclusive execution. In: 2023 60th ACM/IEEE design automation conference (DAC), IEEE, pp 1–6
Lv M, Peng X, Xie W, et al (2022) Task allocation for real-time earth observation service with leo satellites. In: 2022 IEEE real-time systems symposium (RTSS), IEEE, pp 14–26
Melani A, Bertogna M, Bonifaci V, et al (2015) Response-time analysis of conditional DAG tasks in multiprocessor systems. In: 2015 27th Euromicro conference on real-time systems, IEEE, pp 211–221
Melani A, Bertogna M, Bonifaci V et al (2016) Schedulability analysis of conditional parallel task graphs in multicore systems. IEEE Trans Comput 66(2):339–353
Qamhieh M, Fauberteau F, George L, et al (2013) Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In: Proceedings of the 21st international conference on real-time networks and systems, pp 287–296
Sun J, Guan N, Wang Y, et al (2017) Real-time scheduling and analysis of OpenMP task systems with tied tasks. In: 2017 IEEE real-time systems symposium (RTSS), IEEE, pp 92–103
Sun J, Chi Y, Xu T, et al (2020) On the volume calculation for conditional DAG tasks: hardness and algorithms. In: 2020 design, automation & test in Europe conference & exhibition (DATE), IEEE, pp 204–209
Sun J, Guan N, Guo Z, et al (2021) Calculating worst-case response time bounds for OpenMP programs with loop structures. In: 2021 IEEE real-time systems symposium, IEEE, pp 123–135
Sun J, Duan K, Li X, et al (2023) Real-time scheduling of autonomous driving system with guaranteed timing correctness. In: 2023 IEEE 29th real-time and embedded technology and applications symposium (RTAS), IEEE, pp 185–197
Tang Y, Guan N, Yi W (2022) Real-time task models. Handbook of real-time computing. Springer, Cham, pp 469–487
Ueter N, Von Der Brüggen G, Chen JJ, et al (2018) Reservation-based federated scheduling for parallel real-time tasks. In: 2018 IEEE real-time systems symposium (RTSS), IEEE, pp 482–494
Voudouris P, Stenström P, Pathan R (2022) Bounding the execution time of parallel applications on unrelated multiprocessors. Real-Time Syst 2022:1–44

Zhao S, Dai X, Bate I, et al (2020) DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In: 2020 IEEE real-time systems symposium (RTSS), IEEE, pp 128–140

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Qingqiang He is currently a postdoctoral fellow at The Hong Kong Polytechnic University. He received his Ph.D. degree in computer science from The Hong Kong Polytechnic University in 2023. His research interests include real-time scheduling theory and embedded real-time systems. He received the Outstanding Paper Award of IEEE Real-Time Systems Symposium (RTSS) in 2022.

Yongzheng Sun obtained his BE degree from Nanjing University of Science and Technology in 2021 and MSc degree from the Chinese University of Hong Kong in 2022. He is working as a research assistant with the Hong Kong Polytechnic University. His research interests include real-time systems and embedded systems.

Xu Jiang received the BS degree in computer science from Northwestern Polytechnical University, China, in 2009, the MS degree in computer architecture from the Graduate School of the Second Research Institute of China Aerospace Science and Industry Corporation, China, in 2012, and the PhD degree from Beihang University, China, in 2018. Currently, he is working with Northeastern University, China. His research interests include real-time systems, parallel and distributed systems, and embedded systems.

Mingsong Lv received his Ph.D. degree in computer science from Northeastern University, China, in 2010. He is currently with the Hong Kong Polytechnic University. His research interests include timing analysis of real-time systems and intermittent computing.

Jinkyu Lee received the BS, MS, and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2004, 2006, and 2011, respectively. He is currently an associate professor with the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Republic of Korea, where he joined in 2014. He has been a visiting scholar/research fellow with the Department of Electrical Engineering and Computer Science, University of Michigan, USA in 2011–2014. His research interests include system design and analysis with timing guarantees, QoS support, and resource management in real-time embedded systems, mobile systems, and cyber-physical systems. He won the Best Student Paper Award from the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2011, and the Best Paper Award from 33rd IEEE Real-Time Systems Symposium (RTSS) in 2012.

Nan Guan is currently an associate professor at the Department of Computer Science, City University of Hong Kong. Dr. Guan received his BE and MS from Northeastern University, China in 2003 and 2006, respectively, and a Ph.D. from Uppsala University, Sweden in 2013. Before joining CityU, he worked in The Hong Kong Polytechnic University and Northeastern University, China. His research interests include real-time embedded systems and cyber-physical systems. He received the EDAA Outstanding Dissertation Award in 2014, the Best Paper Award of IEEE Real-time Systems Symposium (RTSS) in 2009, the Best Paper Award of Conference on Design Automation and Test in Europe (DATE) in 2013.

Authors and Affiliations

Qingqiang He[1] · Yongzheng Sun[1] · Xu Jiang[2] · Mingsong Lv[1,2] · Jinkyu Lee[3] · Nan Guan[4]

✉ Qingqiang He
qiang.he@connect.polyu.hk

Yongzheng Sun
yongzsun@polyu.edu.hk

Xu Jiang
jiangxu617@163.com

Mingsong Lv
mingsong.lyu@polyu.edu.hk

Jinkyu Lee
jinkyu.lee@skku.edu

Nan Guan
nanguan@cityu.edu.hk

1 The Hong Kong Polytechnic University (PolyU), PQ605, 11 Yuk Choi Rd, Hung Hom, Hong Kong, China

2 Northeastern University, Shenyang, China

3 Sungkyunkwan University, Seoul, South Korea

4 City University of Hong Kong, Kowloon Tong, China