Journal of Systems Architecture 90 (2018) 34–43

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Bounding carry-in interference for synchronous parallel tasks under global fixed-priority scheduling

Meiling Han, Tianyu Zhang, Qingxu Deng*

School of computer science and engineering, Northeastern University, China

A R T I C L E I N F O

Keywords:
Embedded real-time systems
Global fixed-priority
Multiprocessor systems
Response time analysis
Synchronous parallel tasks

A B S T R A C T

With the increasing trend towards using multi-core architecture for embedded systems, the study of intra-task parallelism becomes attractive and desirable in the literature. Although several work studying parallel task models has been proposed, the problem of precise scheduling analysis for the multiprocessor case has largely remained open. To this end, this paper concentrates on analyzing the response time for synchronous parallel real-time tasks scheduled on a multiprocessor platform. Specifically, by exploring the feature of each interfering task, we first present an interference analysis method with higher accuracy compared to other existing work. Considering the cost brought by a high complexity of the proposed method, we further introduce techniques to increase the efficiency with an acceptable loss of accuracy which gives more flexibility to the system designers. Finally, we provide a dynamic programming algorithm for analyzing the schedulability of the whole task set based on our proposed interference analysis technique. Experimental evaluation validates the performance and efficiency of the proposed approach by comparing with other methods.

1. Introduction

Due to the increasing demand of the functionality and service quality, embedded real-time systems are shifting from single-core to multi-core processors. Applications are also required to fully exploit the computation capacity of multiprocessor platforms. To achieve this, designers are committed to design parallelized applications. However, traditional researches on real-time scheduling assume that applications are executed sequentially, which means any particular task is allowed to be executed upon at most one processor at each time instant. Several techniques (e.g., [4,10,13,15,18]) have been proposed to migrate the traditional schedulability analysis techniques to be used for parallelism settings. However, such a migration is non-trivial and several challenges need to be tackled.

Worst case response time analysis for parallel tasks becomes much more complex compared to that for sequentially executed tasks. Recently, there have been several promising techniques developed for parallel tasks schedulability analysis (e.g., [2,3,10,13]) and response time analysis (e.g., [11,17,20]) under global multiprocessor scheduling. In this paper, we study response time analysis for *synchronous parallel tasks* (sp-tasks) under global fixed-priority (G-FP) scheduling. In the sp-task model, each task is composed of a certain number of segments each of which contains several threads. A thread of a segment cannot start executing until all threads of the previous segment have been completed.

The challenge of analyzing the worst case response time of sp-tasks under G-FP scheduling comes from how to upper-bound the total interference of higher priority tasks. Moreover, intra-task parallelism of sp-tasks further aggravates this problem and brings significantly high complexity to the analysis. To tackle this, Melani et al. [20] assumed that the total executions of a task are executing in parallel on all processors. Then, the worst case workload of a parallel task generated in an interval can be computed based on the worst case scenario assumptions for a sequential task. However, such an assumption in [20] may lead pessimistic analysis. Some studies (e.g., [3,10,14,17]) computed the interference of higher priority tasks in a much more precise way and they assumed that all higher priority tasks have carry-in, which is defined as the first instant of a sp-task executed in the interval with release time before and finish time after the beginning of this interval. However, there are totally M processors available for all interfering sp-tasks at each time instant. That is, at most M threads of interfering inter-tasks execute simultaneously just before the beginning of the interval of interest.

In particular, Guan et al. [12] had proved that there are at most $M-1$ interfering tasks with carry-in for sequential task systems. Based on this conclusion, Guan et al. proposed a more precise method to derive an upper bound of interference by departing all interfering tasks into two groups: interfering tasks with carry-in and interfering tasks without carry-in. The interference of each task derived by the method in [12] is much more precise than the interference computed based on the worst case scenario assumption in other methods (e.g., [5,6]). Therefore, the

* Corresponding author.
E-mail address: dengqx@mail.neu.edu.cn (Q. Deng).

https://doi.org/10.1016/j.sysarc.2018.08.004
Received 14 January 2018; Received in revised form 28 June 2018; Accepted 7 August 2018
Available online 8 August 2018
1383-7621/

analyzing methods for sp-tasks based on the assumption that all higher priority tasks have carry-in are significantly pessimistic.

In this paper, we propose a technique to derive an improved upper bound of interference for each interfering inter-task by adopting the technique proposed in [12] for sequential tasks. First, we derive an upper bound of interference for each interfering task with a more precise analysis. Such an improved precision is at the price of higher complexity. To overcome this, we introduce techniques to improve the efficiency of our proposed method. Finally, we provide a dynamic programming algorithm to bound the total interference of tasks with carry-in.

We conduct experiments with randomly generated task sets to evaluate our approach. The results show that our method dominates the one in [17]. We also evaluate the analysis efficiency where the improved approach with higher efficiency can handle large-scale task sets within an acceptable time duration.

1.1. Related work

The real-time community has been devoting significant attention to the problem of scheduling parallel tasks on multiprocessor platforms. One of the mostly common used parallel task model is the fork-join task mode proposed in [15], a fork-join task is composed of an alternating sequence of sequential and parallel segments. The synchronous parallel task model is a special case of the fork-join task model in which a sequential segment is not necessary after a parallel segment. A more general task model is the Directed Acyclic Graph (DAG) task model (e.g., [1,2,16,21]). A DAG task is presented by a directed acyclic graph in which each node represents a thread and each edge between two nodes represents a precedence constraint.

Several EDF-based (Earliest Deadline First) techniques have been proposed to obtain resource augmentation bound for sp-tasks by decomposing each parallel task (e.g., [1,8,16,21]). Most of them break down each parallel task into many smaller sequential tasks. After the decomposing, the set of sequential tasks can be analyzed by using the traditional conclusions for sequential tasks. Some techniques consider the global scheduling policy without decomposing (e.g., [3,11,18]), as federated scheduling (e.g., [2,16]), and RTA (Response Time Analysis) (e.g., [17,18,20]). Recently, Jiang et al. [13] proposed a decomposition-based global EDF scheduling technique for DAG tasks and obtained a sub-optimal result of the capacity augmentation. Chwa et al. [11] proposed a method to analyze the schedulability of sp-tasks based on the concept of at least p-depth interference. The at least p-depth interference means that there are exactly p threads in each counted segment. Such a technique can further reduce the interference of interfering tasks. Recently, the DAG model is extended to the conditional structure (called Conditional DAG model) in [19]. A conditional-DAG model considers the conditional nodes of DAG tasks.

1.2. Organization

The rest of this paper is organized as follows. In Section 2, we introduce the sp-task model and notations used throughout this paper. Next we discuss details of our proposed method in Section 3. Then we introduce how to derive an upper bound of the response time for each sp-task in Section 4. In Section 5, performance of our method is validated using extensive simulation studies. Section 6 concludes the paper.

2. Preliminaries

2.1. Task model

We consider a set of sporadic synchronous parallel tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to be scheduled on a multiprocessor platform under fully preemptive global fixed priority scheduling. The multiprocessor platform is composed of M identical processors. We also assume that tasks in τ are indexed in their priority orders, i.e., τ_i has a higher priority than τ_j if $i < j$. Each task τ_i releases an infinite sequence of jobs. T_i is the minimum inter-release time, which also called the period, of τ_i. D_i is the relative deadline of a task τ_i and $D_i \le T_i$. Each job of τ_i must finish its execution in D_i time units after its release. Without loss of generality, all time intervals and task parameters are assumed to be positive integers.

Table 1
Notations.

Notation	Description
M	The number of processors
τ	$\{\tau_1, \tau_2, \ldots, \tau_n\}$
τ_i	The ith task in the task set
T_i	The period of task τ_i
D_i	The relative deadline of task τ_i and $D_i \le T_i$
$\sigma_{i,j}$	The jth segment of task τ_i and $1 \le j \le s$
$m_{i,j}$	The number of threads of segment $\sigma_{i,j}$
$\Theta_{i,j,z}$	The zth thread in segment $\sigma_{i,j}$ and $1 \le z \le m_{i,j}$
$LC_{i,j,z}$	The WCET of thread $\Theta_{i,j,z}$
$LC_{i,j}$	The maximum WCET of segment $\sigma_{i,j}$
$\Theta^{cp}_{i,j}$	The thread which with the maximum WCET
$C_{i,j}$	The total WCET of segment $\sigma_{i,j}$
LC_i	The length of critical path
C_i	The total WCET of all threads of task τ_i
m_i	The largest degree of parallelism of task τ_i
R_i	The upper bound of response time of task τ_i

Each task τ_i is composed of s segments. A segment of τ_i is denoted by $\sigma_{i,j}$, and consists of $m_{i,j}$ threads ($j \in \{1, 2, \ldots, s\}$). Each thread of $\sigma_{i,j}$ is denoted by $\Theta_{i,j,z}$ ($z \in \{1, 2, \ldots, m_{i,j}\}$) and has a Worst Case Execution Time (WCET) denoted by $LC_{i,j,z}$, see Fig. 1. A segment $\sigma_{i,j}$ can start executing if and only if all threads of the previous segment (if any) have been completed. All threads in a segment are independent and can be executed in parallel. All released threads have no other shared resources except processing units.

The maximum degree of parallelism of task τ_i is denoted by m_i and is defined as $m_i = \max_{j=1}^{s}(m_{i,j})$. This work considers a fully preemptive multi-processor platform, i.e., any executed thread can be preempted and resumed after without any cost. At each instant, the M ready highest priority threads are chosen to be executed on processors.

The WCET of each segment $\sigma_{i,j}$ is captured by the worst case time of finishing all threads of $\sigma_{i,j}$ on a single processor and defined as $C_{i,j} = \sum_{k=1}^{m_{i,j}} LC_{i,j,z}$. Based on this concept, the WCET of τ_i is defined as $C_i = \sum_{j=1}^{s} C_{i,j}$.

When the platform has infinite processors, τ_i is completed after LC_i time units. LC_i is called the critical length of τ_i and defined as $LC_i = \sum_{j=1}^{s} LC_{i,j}$, where $LC_{i,j}$ is the largest WCET of all threads of segment $\sigma_{i,j}$. $LC_{i,j}$ is also called the WCET of segment $\sigma_{i,j}$. Formally, $LC_{i,j} = \max_{z=1}^{m_{i,j}} LC_{i,j,z}$.

A thread $\Theta_{i,j,z}$ of $\sigma_{i,j}$ is called as a critical thread if it is the last one finishing its execution among all threads of $\sigma_{i,j}$ and denoted by $\Theta^{cp}_{i,j}$. Since all threads of $\sigma_{i,j}$ have the same priority, we assume the thread of $\sigma_{i,j}$ with the largest WCET is the critical thread. Obviously, if a task is schedulable, $LC_i \le D_i$ (note that $C_i \le D_i$ is not necessary). We denote the utilization of τ_i as U_i, $U_i = C_i/T_i$. Let U denotes the total utilization of tasks in τ, $U = \sum_{i=1}^{n} U_i$. Clearly, if $U > M$, the task set is not schedulable.

J_i is an arbitrary job of task τ_i with a release time r_{J_i} and a finish time f_{J_i}. Then all the threads of J_i are released at r_{J_i} and the last critical thread is completed at f_{J_i}. J_i is schedulable if $f_{J_i} \le r_{J_i} + D_i$. A task τ_i is schedulable if any job of it is schedulable. The response time of J_i is defined as $R_{J_i} = f_{J_i} - r_{J_i}$. The worst case response time R_i is an upper bound on the response time of all possible jobs of τ_i at run time. If $R_i \le D_i$, this task is schedulable.

For the sake of simplicity, we use the following notation to express that a value A is "limited" if it is bounded by a threshold value similar with [12]. $[\![A]\!]_B = \max(A, B)$, $[\![A]\!]^C = \min(A, C)$, and $[\![A]\!]_B^C = [\![[\![A]\!]_B]\!]^C$. This expression just keeps the value A if it is within the interval $[B, C]$, otherwise it equals to B if $A < B$ or C if $A > C$.

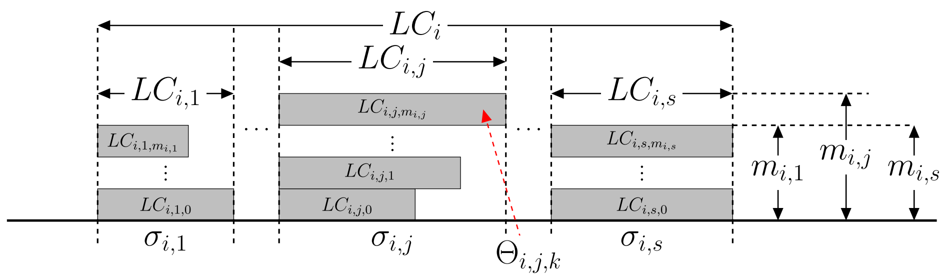


Fig. 1. A sp-task τ_i.

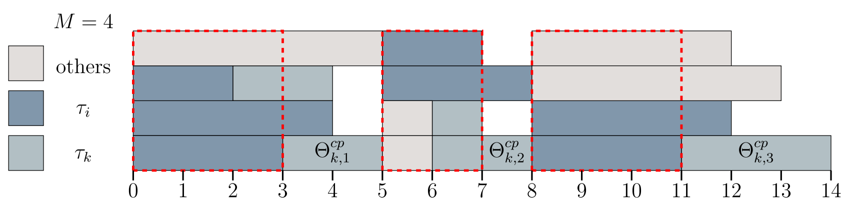


Fig. 2. τ_k only has 3 segments.

2.2. Background

In this section, we review some useful concepts proposed in prior work. Chwa et al. proposed a method to compute the interference of sp-tasks called at least p-depth interference in [11], which is formally defined as follow.

Definition 1. The *at least p-depth critical interference* of task τ_i on τ_k in the interval of interest with length L_k is defined as the total amount of time in which the execution of a critical thread of τ_k is delayed because there are at least p threads of task τ_i simultaneously executing in the system and denoted by $I^p_{i,k}(L_k)$.

The at least p-depth interference is derived from the notion called at least p-depth workload, which is defined in [11] as follow.

Definition 2. The *at least p-depth workload* of a task τ_i in a interval with length L_k is defined as the sum of all intervals in which at least p threads of τ_i are executed simultaneously in parallel and denoted by $W^p_{i,k}(L_k)$.

Example 1. For the analyzed task τ_k which has only 3 segments and $L_k = 14$ shown in Fig. 2, according to the definition of at least p-depth workload, we have $W^1_{i,k}(14) = 11, W^2_{i,k}(14) = 8$ and $W^3_{i,k}(14) = 7$. According to the definition of at least p-depth critical interference, only the workload in the dashed rectangles in Fig. 2 can interfere with τ_k.

Note that when the at least p-depth workload of τ_i is large enough and a portion of it has to be executed in parallel with the critical threads of τ_k, see Fig. 2, the parallel executing portion of τ_i cannot contribute to interfering with the execution of τ_k. Then the following equation holds:

$$I^p_{i,k}(L_k) \le [\![W_{i,k}{}^p(L_k)]\!]^{L_k - LC_k + 1}.$$

Note that the upper bound of the at least p-depth workload of τ_i is $L_k - LC_k + 1$ rather than $L_k - LC_k$, to facilitate the iterative response time analysis procedure. A formal explanation of this issue can be found in [6].

The critical interference of τ_i on τ_k in the interval with length L_k is defined as the sum of at least p-depth interference where $1 \le p \le m_i$ and denoted by $I_{i,k}(L_k)$. Formally,

$$I_{i,k}(L_k) = \sum_{p=1}^{m_i} I^p_{i,k}(L_k). \tag{1}$$

For example, the execution situation shown in Fig. 2, the $I_{i,k}(L_k) = 8 + 8 + 7 = 23$.

We assume an arbitrary job J_k of τ_k is the analyzed job. Each critical thread of J_k is interfered by threads of higher priority tasks and the threads of J_k which are not the critical threads. We call a thread as an *interfering thread* if it interferes with the execution of critical threads of J_k.

Following a typical approach adopted in the response time analysis for globally scheduled systems [12], we divide the workload of an interfering task τ_i in the interval of interest with length L_k to carry-in, body and carry-out jobs (see Fig. 4). The carry-in job is the first instance of τ_i executing in the interval with release time before and deadline within the interval. The carry-out job is the last instance of τ_i in the interval having release time within and deadline after the interval. The rest jobs of τ_i in the interval are called body jobs. The *interfering threads* of τ_i are defined as the threads which are executed in the interval of interest. The *interfering threads* of τ_i can be classified into three types: carry-in, body and carry-out threads, depending on which kind of jobs they belong to. The interval of interest is the busy period [5] of an analyzed task τ_k and defined as follow:

Definition 3. The busy period of τ_k is defined as the interval in which all processors are busy with executing the interfering threads or the critical threads of τ_k.

3. Overview

In this section, we give an overview of our method proposed in this work. We first clarify the pessimistic of the existing work on response time analysis for parallel tasks and explain the challenge to derive an accurate upper bound on the interference of each interfering parallel task. Then, we discuss how to extend the technique in [12] to be adoptable for the sp-task model. In the following, we use τ_k to denote the analyzed task.

3.1. Pessimistic of the existing work

In most existing work which analyzes the schedulability for sp-tasks, the interference of a task τ_i is computed by the initial scenario. The initial scenario for sp-tasks is found in the situation where (i) the carry-in job are executed as late as possible and its execution start time is aligned with the beginning of the interval of interest, and (ii) later jobs are executed as soon as possible. Note that the assumptions of the initial scenario for sp-tasks are the same as that for sequential tasks in [6]. However, for sp-tasks, such an assumption may not be able to derive the

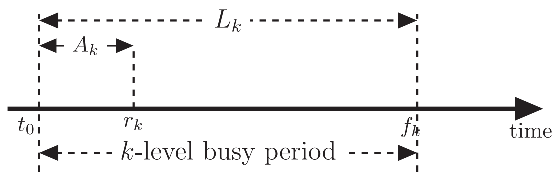


Fig. 3. Busy period of τ_k.

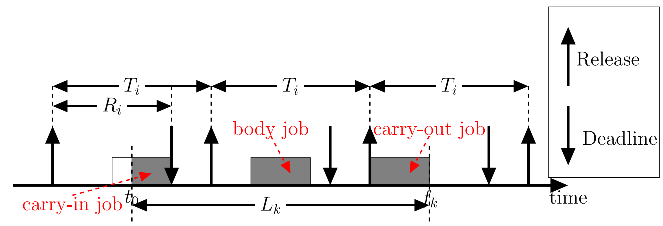


Fig. 4. The composition of L_k for task τ_i.

worst case interference for each interfering task τ_i, due to the following reasons [20]:

(1) If we slide the interval of interest to left, a larger workload may happen in the interval, due to the precedence constraints and the different degree of parallelism of each segment of τ_i. A greater workload may occur if the slide-out segment of the carry-out job has a smaller degree of parallelism than the slide-in segment of the carry-in job.

(2) A sustainable schedulability analysis [9] must guarantee that all tasks meet their deadlines even when some of them are executed less than their WCETs. Assume that a segment $\sigma_{i,j}$ of the carry-out job of τ_i has a less degree of parallelism than $\sigma_{i,j+1}$, and the former one is executed in and the later one is executed out the interval of interest. If $\sigma_{i,j}$ is executed less than $LC_{i,j}$, segment $\sigma_{i,j+1}$ might start its execution in the interval of interest.

To derive a safe upper bound of interference of all interfering tasks, a straightforward method is enumerating all execution patterns of the carry-in and carry-out jobs. Obviously, this will lead to horrible efficiency problem. To reduce the complexity, Maia et al. [17] proposed a sliding technique to find the worst case situation by re-decomposing the carry-out job. Such a technique resorts segments of a carry-out job in the descending order of their degrees of parallelism. Combining with the initial scenario introduced before, they derived an upper bound of the interference of each interfering task.

Note that the method in [17] is based on an over-estimation on the interference of tasks with carry-in. They assumed that all higher priority tasks have carry-in and the initial state of sliding procedure for each sp-task is that the carry-in job starting execution aligned with the beginning of the interval of interest. But for sequential tasks, the worst case scenario has been updated by a more precise method proposed by Guan et al. [12]. In the next section, we present how to extend the busy period extending theory used in [12] to sp-tasks to bound the carry-in interference.

3.2. Worst case scenario

Assume that an sp-task τ_k is released at r_k. The execution of τ_k's critical threads is delayed while the interfering threads are executing. Note that the delay only occurs when all processors are busy with interfering threads. According to the busy period extension idea proposed in [12], we extend the starting time of the interval of interest from r_k to an earlier time instant t_0, which is defined as the earliest time instant before r_k and at any time instant $t \in [t_0, r_k)$ all processors are busy with interfering threads. If no such a time instant exists, we set $t_0 = r_k$. The busy period of τ_k starts at t_0 and ends at f_k which is the finish time of the last critical thread of τ_k. We define $A_k = r_k - t_0$ (see Fig. 3). Window $[t_0, f_k)$ is denoted as the *k-level busy period*. At time instant $t_0 - 1$, there are strictly less than M processors being busy with interfering threads. We state this property as the following lemma.

Lemma 1. *At $t_0 - 1$, there are at most $M - 1$ threads of interfering tasks being executed, which are released before t_0 and finished after t_0.*

Proof. According to the discussion above, the proof can be constructed in a similar way with [12]. □

Based on Lemma 1, there are at most $M - 1$ interfering threads being executed at $t_0 - 1$, and we define these threads as *fore-carry-in* threads.

Definition 4. A thread is a *fore-carry-in* thread, if it is a carry-in thread and being executed at $t_0 - 1$.

A higher priority sp-task τ_i has carry-in, if τ_i has at least one fore-carry-in thread. τ_i has no carry-in, if a higher priority sp-task has no fore-carry-in thread. Based on such a classification, the worst case scenario of an interfering task τ_i should be discussed by two cases depending on whether it has carry-in.

We discuss the worst case scenario for interfering task τ_i without carry-in in the k-level busy period. The length of the k-level busy period equals L_k and it starts at t_0. Since an sp-task τ_i has no carry-in, the number of body jobs that can be executed in the k-level busy period is the greatest when it releases its first job at t_0 and its later jobs as soon as possible. Due to the second reason discussed in Section 3.1, we use the re-decomposing technique in [17] to derive an upper bound on interference of τ_i when it has no carry-in. Finally, the worst case scenario for τ_i can be found in the situation where (i) τ_i releases its first job at t_0, (ii) the later jobs are released as soon as possible,and (iii) all threads are executed in their WCET and the segments of carry-out job are re-decomposed in descending order of their degrees of parallelism.

To specify the workload of carry-in job and carry-out job in a given interval, we define the carry-in window and carry-out window as follows.

Definition 5. The time interval which is a part of a busy period and in which a carry-in job is executed is defined as a carry-in window. Similar, the interval which is a part of busy period and in which a carry-out job is executed is defined as a carry-out window.

Note that a carry-in and carry-out window are changed for different tasks in a same busy period. The number of body jobs is denoted by $N_i^{nc}(L_k)$ and the length of the carry-out window is denoted by $X_i^{cout}(L_k)$. In the worst case scenario discussed above, they can be computed by the following lemma.

Lemma 2. *The number of body jobs and the length of carry-out window of τ_i when it has no carry-in can be computed by (2) and (3), respectively.*

$$N_i^{nc}(L_k) = \left\lfloor \frac{L_k}{T_i} \right\rfloor \tag{2}$$

$$X_i^{cout}(L_k) = [\![L_k \ mod \ T_i]\!]^{LC_i} \tag{3}$$

Proof. According to the above discussion, the proof can be constructed in a similar way with [12]. □

Now we discuss the worst case scenario of each interfering task when it has carry-in. According to Lemma 1, there are at most $M - 1$ fore-carry-in threads in the worst case scenario of the execution of τ_k. Then the problem of deriving an upper bound on interference of all tasks with carry-in in the k-level busy period can be described as follows. There is a bag with a size of $M - 1$ and there are n_{hp} items. Each item τ_i has a size with a value from 1 to $M - 1$. Each size ω_i of τ_i corresponds to a value denoted by $V_i^{\omega_i}$. n_{hp} is the number of tasks with higher priority and ω_i is the number of fore-carry-in threads of τ_i. $V_i^{\omega_i}$ denotes the interference of τ_i when it has ω_i fore-carry-in threads. τ_{ch} denotes the set of items which have been selected to fill the bag with size $M - 1$. Obviously, the computation of an upper bound interference of all higher priority sp-tasks with fore-carry-in threads is reduced to computing the maximum $\sum_{\tau_i \in \tau_{ch}} V_i^{\omega_i}$ under condition (4).

$$\sum_{\tau_i \in \tau_{ch}} \omega_i \leq M - 1 \tag{4}$$

Computing the upper bound interference of tasks with carry-in is reduced to a multi-dimensional knapsack problem and can be resolved by a dynamic programming algorithm. We will present the algorithm in the following section. To be clear, we give a summary definitions of symbols used above as following:

- τ_i is a higher priority task with carry-in.
- ω_i is defined as the number of fore-carry-in threads of τ_i and $\omega_i \in [1, [\![M-1]\!]^{m_i}]$.
- $V_i^{\omega_i}$ is defined as the worst case interference of τ_i when it has ω_i fore-carry-in threads in the k-level busy period.

Assume the length of carry-in window is X_i^{ci}. To upper-bound the interference generated by the carry-in job of τ_i when it has ω_i fore-carry-in threads, we need to determine which segments of τ_i may be executed within the carry-in window $[t_0, t_0 + X_i^{ci}]$, either fully or partially. Intuitively, to maximize the interference, the carry-in job should be executed as much workload as possible, also, as late as possible. According to the conclusion in [17], the execution pattern of the carry-in job causes the maximum workload in the k-level busy period when it is executed as there are unlimited number of processors and all threads of it is executed in their WCETs as late as possible. Contrary to the carry-in job, the maximum interference generated by the carry-out job in the carry-out widow with length X_i^{cout} can be found when it re-decomposes the order of segments in the descending order of their degrees of parallelism and starts executing as soon as it is released and each segment is executed at the highest degree of parallelism.

Due to the reasons discussed in Section 3.1, we must consider all possible values of the length of the carry-in window to upper-bound the interference of τ_i when it has ω_i fore-carry-in threads. However, the changeable carry-in window can only affect the length of the carry-out window but cannot affect the greatest number of body jobs of τ_i in the k-level busy period. Then the greatest number of body jobs of τ_i when it has carry-in can be derived according to the worst case assumption for sequential tasks (see Fig. 4). The worst case scenario assumed in [12] for a sequential task with carry-in is found in the situation where: (i) the carry-in job is executed in its worst case scenario, (ii) the later jobs are released with the minimum separation interval, and (iii) the carry-out job is finished just at the end of the interval of interest. The greatest number of body jobs of τ_i can be computed by:

$$N_i^{ci}(LC_k) = [\![\left\lfloor \frac{L_k - LC_i}{T_i} \right\rfloor]\!]_0 .$$

According to the discussion above, the total length of $X_i^{ci} + X_i^{cout}$ of τ_i in the k-level busy period with length L_k is fixed and can be computed by the following lemma.

Lemma 3. *In the k-level busy period with length L_k, the total length of $X_i^{ci} + X_i^{cout}$ of τ_i is bounded by X_i^{in} which is computed by (5).*

$$X_i^{in} \leq [\![L_k - N_i^{ci}(LC_k) \times T_i - [\![T_i - R_i]\!]_0]\!]_0 + LC_i \tag{5}$$

Proof. According to the discussion above, the lemma is proved. □

4. Response time analysis

In this section, we describe how to derive an upper bound on the worst case response time of each sp-task, based on the worst case scenario discussion in Section 3.2. The main challenge of the response time analysis is to derive an upper bound on the interference of all interfering tasks, which is caused by threads of higher priority tasks and threads of itself except critical threads. The interference from higher priority tasks is called inter-task interference and the interference of itself is called intra-task interference. In the following, we assume that any sp-task τ_i interferes the execution of τ_k in the k-level busy period with length L_k. Based on the at least p-depth interference definition, the intra-task interference can be computed by (6).

$$I_{k,k} = \sum_{p=1}^{m_k} [\![\sum_{\forall j: m_{k,j} \geq (p+1)} LC_{i,j}]\!]^{L_k - LC_k + 1} \tag{6}$$

Next we introduce how to compute the upper bound interference of inter-tasks.

4.1. Inter-task interference

4.1.1. Interference of an sp-task without carry-in

The worst case interference from a higher priority task τ_i without carry-in is composed of two parts: the interference of body jobs and the interference of the carry-out job. According to Definition 1, the at least p-depth workload of one body job is denoted by $\beta_{i,k}^p(p, L_k)$ and can be computed by (7). The length of the carry-out of τ_i is defined as X_i^{nc} and can be computed by (3). Then, the at least p-depth workload of the re-decomposed carry-out job, denoted by $\gamma_{i,k}^p(J_i^{de}, p, X_i^{nc})$, can be computed by (8).

$$\beta_{i,k}^p(p, L_k) = \sum_{\forall j: m_{i,j} \geq p} LC_{i,j} \tag{7}$$

$$\gamma_{i,k}^p(J_i^{de}, p, X_i^{nc}) = \begin{cases} 0 & \text{if } X_i^{nc} <= 0, \\ \sum_{j=1, m_{i,j} >= p}^{z} LC_{i,j} + (X_i^{nc} - \sum_{j=1}^{z} LC_{i,j}) & \text{if } 0 < X_i^{nc} <= LC_i \text{ and } m_{i,z+1} >= p, \\ \sum_{j=1, m_{i,j} >= p}^{z} LC_{i,j} & \text{if } 0 < X_i^{nc} <= LC_i \text{ and } m_{i,z+1} < p, \\ \sum_{\forall j; m_{i,j} >= p} LC_{i,j} & \text{otherwise} \end{cases} \tag{8}$$

where, z is the index of the last segment that is fully included in the busy window, and J_i^{de} denotes the re-decomposed carry-out job.

By adding the at least p-depth workload of the body and carry-out jobs, the worst case at least p-depth workload of interfering τ_i when it has no carry-in is defined by $W_{i,k}^{nc,p}(p, L_k)$ and can be computed by (9).

$$W_{i,k}^{nc,p}(p, L_k) = N_i^{nc}(L_k) \times \beta_{i,k}^p(p, L_k) + \gamma_{i,k}^p(J_i^{de}, p, X_i^{nc}) \tag{9}$$

According to Definition 1 and (9), the total interference of τ_i on τ_k in the k-level busy period with length L_k can be computed by (10).

$$I_{i,k}^{nc}(\tau_i, L_k) = \sum_{p=1}^{m_i} [\![W_{i,k}^{nc,p}(p, L_k)]\!]^{L_k - LC_k + 1} \tag{10}$$

4.1.2. Interference of sp-tasks with carry-in

In this section, we discuss how to compute the worst case interference of τ_i on τ_k in the k-level busy period when τ_i has ω_i fore-carry-in threads. The start time of the k-level busy period is t_0 and the length of it is L_k. The total length of carry-in and carry-out is defined as $X(L_k)$, which is computed by (5). By enumerating all possible values of X_i^{ci}, we can find the worst case value that derives the largest interference of τ_i when it has ω_i fore-carry-in threads. Note that the interference contribution has a discontinuity whenever one of the extreme points of the busy period coincides with a segment boundary. That is, the worst case interference happens when (i) t_0 is aligned with a segment boundary of the carry-in job, or (ii) f_k is aligned with a segment boundary of the carry-out job.

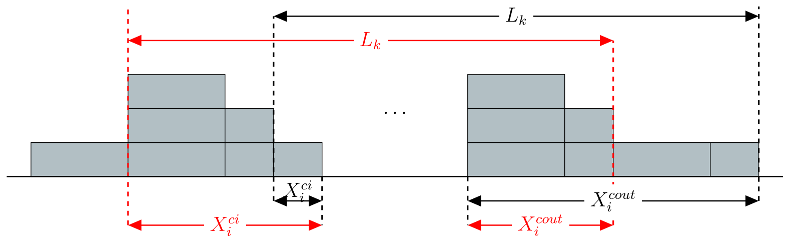


Fig. 5. Illustration of enumerate all possible cases of X_i^{ci}.

X_i^{ori} is defined as the carry-in length for task τ_i in the busy period $[t_0, f_k)$ of τ_k when its carry-out job is finished at f_k.

$$X_i^{ori} = [\![[\![(L - LC_i) \ mod \ T_i - [\![T_i - R_i]\!]_0]\!]_0]\!]^{LC_i} \tag{11}$$

In the first case, if X_i^{ci} is determined, X_i^{cout} is also fixed. Γ'_{ci} denotes a set of all possible values of X_i^{ci} in this case. Note that the meaningful length of the carry-in window for each task τ_i happens when the start time of the k-level busy period is aligned with the start time of each segment $\sigma_{i,j}$ of τ_i's carry-in job correspondingly. Moreover, X_i^{ci} cannot be greater than LC_i and be less than X_i^{ori} since X_i^{cout} cannot be less than 0 and be greater than LC_i. Therefore, we have

$$\Gamma'_{ci} = \left\{ X_i^{ci} = \sum_{j=s-\Delta}^{s} LC_{i,j} \middle| X_i^{ori} \le X_i^{ci} \le LC_i, \Delta \in 0, \dots, s-1 \right\}. \tag{12}$$

In the second case, if X_i^{cout} is determined, X_i^{ci} is also fixed. Γ'_{cout} denotes the set of all possible values of X_i^{cout}. The meaningful length of the carry-out window for each task τ_i happens when the end time of the k-level busy period corresponds to the end time of each segment $\sigma_{i,j}$ of τ_i's carry-out job. Since $X_i^{ori} \le X_i^{ci} \le LC_i$, $[\![X_i^{in} - LC_i]\!]_0 \le X_i^{cout} \le LC_i$. For simplicity, we define $\varphi_i = [\![X_i^{in} - LC_i]\!]_0$. Therefore, we have

$$\Gamma'_{cout} = \left\{ X_i^{cout} = \sum_{j=1}^{s} LC_{i,j} \middle| \varphi_i \le X_i^{cout} \le LC_i \right\}. \tag{13}$$

Γ''_{ci} denotes the set of X_i^{ci} and each value in it is derived from Γ'_{cout} by the following equation:

$$\Gamma''_{ci} = \left\{ X_i^{ci} = X_i^{in} - X_i^{cout} \middle| \varphi_i \le X_i^{ci} \le LC_i, X_i^{cout} \in \Gamma'_{cout} \right\}.$$

We use Γ_{ci} to denote the set of all possible values of X_i^{ci} for task τ_i. Formally,

$$\Gamma_{ci} = \Gamma'_{ci} \cup \Gamma''_{ci}.$$

Each value X_i^{ci} in Γ_{ci} corresponds to a value of X_i^{cout} and is computed by the following equation.

$$X_i^{cout} = [\![X_i^{in} - X_i^{ci}]\!]_{\varphi_i}^{LC_i}.$$

Example 2. For example, a task $\tau_{i<k}$ executed in a k-level busy period with length L_k shown in Fig. 5 in black lines, this interval is an original interval. In this case, the X_i^{ci} is the WCET of the last segment and the X_i^{cout} is the length of the length of the critical path. When sliding the k-level busy period to left in red lines, still with length L_k, we got a new length of X_i^{ci} the length of the last three segments and X_i^{cout} which reduced by the length increasing in the carry-in window. After changing the length of carry-in, the task τ_i obviously contributes more interference than the first case.

By enumerating all values in Γ_{ci}, we can find the worst case interference for τ_i when it has ω_i fore-carry-in threads. In this combination, we assume that the ω_i fore-carry-in threads belong to the segment $\sigma_{i,j}^{ci}$. If $m_{i,j}^{ci} < \omega_i$, the worst case interference of this case equals to the case when there are $m_{i,j}^{ci}$ fore-carry-in threads. If $m_{i,j}^{ci} > \omega_i$, some threads of $\sigma_{i,j}^{ci}$ should be assumed as being executed with 0 to make the number of fore-carry-in threads equal with ω_i. We denote the re-defined carry-in job as $J_i^{\omega_i}$, and the at least p-depth workload of the carry-in job with length X_i^{ci} is computed by (14). We denote the at least p-depth workload of $J_i^{\omega_i}$ with length X_i^{ci} as $\alpha_{i,k}^p(J_i^{\omega_i}, p, X_i^{ci})$.

$$\alpha_{i,k}^p(J_i^{\omega_i}, p, X_i^{ci}) = \begin{cases} 0 & \text{if } X_i^{ci} <= 0 \\ \sum\limits_{j=h, m_{i,j}>=p}^{s} LC_{i,j} + \left(X_i^{ci} - \sum\limits_{j=h}^{s} LC_{i,j} \right) & \text{if } 0 < X_i^{ci} <= LC_i \text{ and } m_{i,h-1} >= p, \\ \sum\limits_{j=h, m_{i,j}>=p}^{s} LC_{i,j} & \text{if } 0 < X_i^{ci} <= LC_i \text{ and } m_{i,h-1} < p, \\ \sum\limits_{\forall j; m_{i,j}>=p} LC_{i,j} & \text{otherwise.} \end{cases} \tag{14}$$

where, h denotes the earliest segment that is fully included in an analyzed window with length L_k.

Note that for a task τ_i which has ω_i fore-carry-in threads, we need to compute $|\Gamma_{ci}|$ times to find the worst case interference for deriving an upper bound on total interference of all tasks with carry-in. $|\Gamma_{ci}|$ is defined as the number of elements in set Γ_{ci}. If the number of values of ω_i is n_{ω_i}, we need to compute $n_{\omega_i} \times |\Gamma_{ci}|$ times to obtain all legal values of $\alpha_{i,k}^p(J_i^{\omega_i}, p, X_i^{ci})$ to find the worst case interference of τ_i for all cases of ω_i. This can cause significant efficiency problem. To tackle this, we propose a method to reduce the complexity of this procedure.

We re-decompose the structure of the carry-in job by an increasing order of degrees of parallelism and denoted by J_i^{as}(see Fig. 6). If we do not consider the number of fore-carry-in threads for task τ_i, we can compute the worst case workload of τ_i in the k-level busy period when $X_i^{ci} = X_i^{wc}$. We assume the first segment of J_i^{as} executed in carry-in window is denoted by $\sigma_{i,j}$. The following lemma is true.

Lemma 4. *When $X_i^{ci} = X_i^{wc}$, if the start time of the k-level busy period is aligned with the boundary of a segment of J_i^{as}, the worst case workload of τ_i for each value of ω_i equals the worst case scenario $X_i^{ci} = X_i^{wc}$.*

Proof. This lemma obviously true, because the changing of the WCETs of the segment $\sigma_{i,j-1}$ cannot effect the workload in the busy period. □

If the beginning of the k-level busy period is not aligned with a segment boundary of the carry-in job, the maximum workload can be found by checking the following lemma:

Lemma 5. *If the beginning of the k-level busy period is not aligned with a segment boundary of the carry-in job (see Fig. 6), the maximum workload is the maximum one among the following cases:*

(1) *$X_i^{ci} = X_i^{wc}$, computing the workload of J_i^{as} and J_i^{de} by adjusting $m_{i,j}$ to ω_i;*
(2) *$X_i^{ci} = X_i^{wc} + X_i^{bt}$, computing the workload of J_i^{as} and J_i^{de} with the increasing value of X_i^{ci}.*
(3) *$X_i^{ci} = X_i^{wc} - X_i^{at}$, computing the workload of J_i^{as} and J_i^{de} with the decreasing value of X_i^{ci}.*

where X_i^{bt} is defined as the interval in which threads of $\sigma_{i,j}^{ci}$ are executed before t_0 and X_i^{at} is defined as the interval in which threads of $\sigma_{i,j}^{ci}$ are executed after t_0.

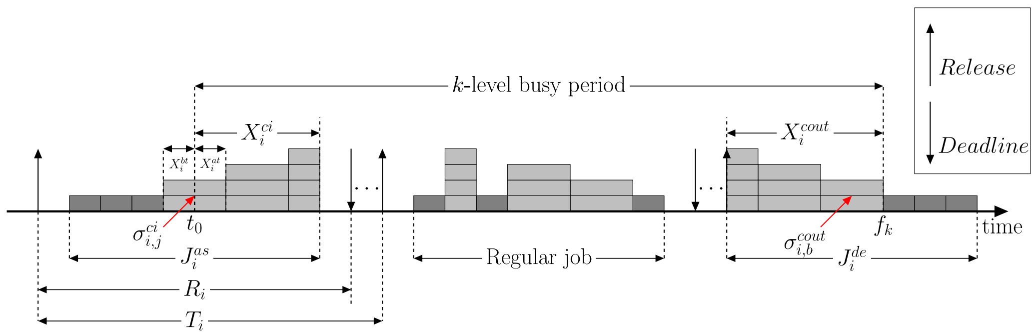


Fig. 6. An sp-task τ_i.

Proof. Since the workload of the body jobs are the same for different values of X_i^{ci}, we only need to consider the varying workload of J_i^{as} and J_i^{de}. J_i^{as} is sorted in an increasing order of degrees of parallelism and J_i^{de} is sorted in a decreasing order (see Fig. 6). To prove this lemma, we should proof for any cases of length X_i^{ci} cannot increasing the workload in the k-level busy period except the three cases. Reducing and increasing the length of X_i^{ci} are corresponding to the cases (2) and (3). So it is sufficient to prove the following two cases:

First, we prove that we cannot obtain a larger workload comparing to the case $X_i^{ci} = X_i^{wc} + X_i^{bt}$. To clarify the reasons, we assume that $\sigma_{i,j}^{ci}$ is the last segment of J_i^{as} with a start time before t_0 when $X_i^{ci} = X_i^{wc}$. We also assume that $\sigma_{i,b}^{cout}$ is the first segment of J_i^{de} with an end time at or after $t_0 + f_k$ when $X_i^{ci} = X_i^{wc}$. Since the worst case workload happens when $X_i^{ci} = X_i^{wc}$ if we do not consider the number of fore-carry-in threads, we have $m_{i,j-1}^{ci} \le m_{i,b}^{cout}$ and $m_{i,b+1}^{cout} \le m_{i,j}^{ci}$ when $1 < j < s_i$ and $1 < b < s_i$. Due to the structure of J_i^{as}, increasing X_i^{ci} can make one or more segments (if any) before $\sigma_{i,j}^{ci}$ with a smaller degree of parallelism than $m_{i,j}^{ci}$ is executed after t_0. Since X_i^{ci} is increased, X_i^{cout} is decreased which makes one or more segments (if any) with a larger degree of parallelism before $\sigma_{i,b}^{cout}$ starts executing after f_k. Obviously, the total workload of J_i^{as} and J_i^{de} is non-increasing with the increasing of X_i^{ci} when $X_i^{ci} \ge X_i^{wc} + X_i^{bt}$. If the worst case scenario does not happen in $X_i^{ci} = X_i^{wc} + X_i^{bt}$, it cannot happen in $X_i^{ci} > X_i^{wc} + X_i^{bt}$ when τ_i has ω_i fore-carry-in threads (see Fig. 6).

Second, we prove that we cannot obtain a larger workload comparing to the case $X_i^{ci} = X_i^{wc} - X_i^{at}$. If the worst case scenario does not happen in $X_i^{ci} = X_i^{wc} - X_i^{at}$, it cannot happen in $X_i^{ci} < X_i^{wc} - X_i^{at}$ when τ_i has ω_i fore-carry-in threads. Because decreasing X_i^{ci} does not make the total workload of J_i^{as} and J_i^{de} become larger than the total workload of J_i^{as} and J_i^{de} when $X_i^{ci} = X_i^{wc} - X_i^{at}$. □

According to the discussion above, we just need to compute $|\Gamma_{ci}| + 3 \times n_{\omega_i}$ times to obtain all legal values of $\alpha_{i,k}^{p}(J_i^{\omega_i}, p, X_i^{ci})$. The time complexity is reduced from $O(n \times (M-1))$ to $O(n + 3 \times (M-1))$.

For a given value of X_i^{ci}, we get a value of X_i^{cout}. We can compute the at least p depth workload of the carry-in window by (14) and the at least p depth workload of the carry-out window by (8). Finally, the at least p depth workload of τ_i in the k-level busy period with length L_k is composed by three parts: the at least p depth workload of carry-in, body and carry-out jobs, for a given value of X_i^{ci}. Formally, we have

$$W_{i,k}^{ci,p}(p, L_k) = \alpha_{i,k}^{p}(J_i^{\omega_i}, p, X_i^{ci}) + N_i^{ci}(LC_k) \times \beta_{i,k}^{p}(p, L_k) + \gamma_{i,k}^{p}(J_i^{de}, p, X_i^{cout}) \tag{15}$$

And the total interference of τ_i with carry-in is computed by (16), when the length of the carry-in window is X_i^{ci} and there are ω_i threads being executed at $t_0 - 1$.

$$I_{i,k}^{ci,\omega_i}(\tau_i, L_k) = \sum_{p=1}^{mi} [\![W_{i,k}^{ci,p}(p, L_k)]\!]^{L_k - LC_k + 1} \tag{16}$$

So far, we have analyzed the worst case interference of τ_i when it has ω_i fore-carry-in threads. Based on this, we will discuss how to derive the upper-bound on the total interference of inter-tasks in the next section.

4.1.3. Total interference of inter-task

We now define the total interference $\Omega_k(L_k)$ as the maximal total interference of all inter-tasks.

$$\Omega_k(L_k) = \max_{(\tau_{nc}, \tau_{ci}) \in Z} \left(\sum_{\tau_i \in \tau_{nc}} I_{i,k}^{nc}(\tau_i, L_k) + \sum_{\tau_i \in \tau_{ci}} I_{i,k}^{ci,\omega_i}(\tau_i, L_k) \right) \tag{17}$$

where $Z \subseteq \tau \times \tau$ is the set of all partitions of the set $\tau_{<k}$ into τ_{nc} and τ_{ci}, such that $\tau_{nc} \bigcap \tau_{ci} = \emptyset$. $\omega_i \in [1, [\![M-1]\!]^{mi}]$. Note that the sum of ω_i of each task τ_i in τ_{ci} is not greater than $M-1$. The partition of τ_{nc} and τ_{ci} with maximum total interference is the one deriving an upper bound on the interference of all inter-tasks. Naturally, enumerating all partitions falls into huge complexity. To overcome this, authors in [12] proposed a concept called *Idiff* of τ_i which is defined as the difference interference between τ_i with carry-in and without carry-in. Then, they can find the $M-1$ largest differences in linear time. We utilize the *Idiff* concept for analyzing the sp-tasks.

Definition 6. The difference interference of τ_i between the case having ω_i fore-carry-in threads and having no carry-in is computed by:

$$d_i^{\omega_i} = I_{i,k}^{ci,\omega_i}(\tau_i, L_k) - I_{i,k}^{nc}(\tau_i, L_k). \tag{18}$$

Then, the total interference of inter-tasks can be redefined by:

$$\Omega_k(L_k) = \sum_{\tau_i \in \tau_{hp}} I_{i,k}^{nc}(\tau_i, L_k) + d_{\sum}(L_k) \tag{19}$$

where $d_{\sum}(L_k)$ denotes the maximum sum of *Idiffs* of tasks with carry-in and the number of fore-carry-in threads is at most $M-1$. The problem of upper-bounding the interference of inter-tasks with carry-in is reduced to deriving the maximum total *Idiffs* of tasks with carry-in and the number of fore-carry-in threads is at most $M-1$. According to the worst case scenario discussed in Section 3, the problem of deriving the maximum total *Idiffs* of tasks with carry-in is a multi-dimensional knapsack problem and can be computed by a dynamic programming algorithm, see Algorithm 1.

In Algorithm 1, a higher priority task τ_i has two parameters. One is a set of values of ω_i denoted by W_i and the other one is a set of values of $d_i^{\omega_i}$ denoted by V_i. Note that each value in W_i corresponds to a value of $d_i^{\omega_i}$ in V_i. *d_set* is a set of (W_i, V_i) for each higher priority task, and we use *item* to denote the index of a inter-task in *d_set* and all inter-tasks are sorted by their priorities. The number of fore-carry-in threads is at most $M-1$, which is called the size of a bag in the following. The procedure of Algorithm 1 can be described as filling a table denoted as *Mv*. *Mv*[*item*][*m*] denotes the maximum interference when the size of the bag is *m* and all tasks have been considered before *item* + 1 in *d_set*.

We take *Mv*[*item*][*m*] as an example. When the size of the bag is *m*, we need to consider whether *d_set*[*item*] can be put in the bag. If we put it in the bag, there are $|W_{item}|$ choices where $|W_{item}|$ is the number

Algorithm 1 Get the maximum sum of all d_Σ.

Require: $M-1, d\_set$

Ensure: the sum of ω_{item} of each chosen item is at most $M-1$ and the sum of $d_i^{\omega_{item}}$ is greatest.

```
1:  function Get_WC(M − 1, d_set)
2:      Mv = [ ]
3:      for ∀m ∈ [1, M − 1] do
4:          for ∀item ∈ [0, len(d_set)] do
5:              W_item ← d_set[item][0]
6:              V_item ← d_set[item][1]
7:              if min(W_item) > m then:
8:                  if item == 0 then
9:                      Mv[item][m] ← 0
10:                 else
11:                     Mv[item][m] ← Mv[item − 1][m]
12:                 end if
13:             else
14:                 if item == 0 then:
15:                     for ∀t_wt ∈ [1, m] do
16:                         V_tmp ← value[t_wt]
17:                     end for
18:                     Mv[item][m] ← max(V_tmp)
19:                 else
20:                     for ∀t_wt ∈ [1, m] do
21:                         if t_wt ≤ m then
22:                             lf_wt ← m − t_wt
23:                             iteminbag = iteminbag ∪ (Mv[item −
    1][lf_wt] + value[t_wt])
24:                         end if
25:                     end for
26:                     itemIn_fn ← max(iteminbag)
27:                     if Mv[item − 1][m] > itemIn_fn then
28:                         Mv[item][m] = Mv[item − 1][m]
29:                     else
30:                         Mv[item][m] = itemIn_fn
31:                     end if
32:                 end if
33:             end if
34:         end for
35:     end for
36:     max_sum = max(Mv[len(d_set) − 1])
37:     return max_sum
38: end function
```

of values in W_{item}. If we chose $d\_set[item]$ with $m = W_{item}[s]$ form W_{item}, which corresponds to $V_{item}[s]$, the size for the items had been put in the bag is denoted by lf_wt, and the maximum sum of interference of them is $Mv[item-1][lf\_wt]$. If we put $V_{item}[s]$ in this bag, the sum of values is updated by $Mv[item-1][lf\_wt] + V_{item}[s]$. Then, the maximum sum of interference is obtained by enumerating all values in W_{item} of $d\_set[item]$ (lines 20–40). If $d\_set[item]$ cannot be put in the bag or:

$$\max_{s \in W_{item}} Mv[item-1][lf\_wt] + V_{item}[s] < Mv[item-1][m]$$

we have $Mv[item][m] = Mv[item-1][m]$ (see lines 26–30).

The result returned by Algorithm 1 is $d_\Sigma(L_k)$. Based on the total interference of inter-tasks and the intra-task, we give a method to derive an upper bound of response time of a task τ_k.

4.2. RTA procedure

In this section, we derive a safe upper bound response time of τ_k in the k-level busy period by combining the worst case response time analysis procedure proposed in [12]. According to the discussion above, the upper bound of the total interference to an sp-task τ_k in the k-level busy period with length L_k is denoted by $\Psi(L_k)$ and defined as:

$$\Psi_k(L_k) = \Omega_k(L_k) + I_{k,k}. \tag{20}$$

Now we describe how to use $\Psi(L_k)$ to conduct the response time analysis for τ_k. As introduced in Section 3, the k-level busy period begins at time instant t_0, which is A_k time units before r_k. r_k is the release time of τ_k. In general, A_k is an open variable. Authors in [12] proved that if the interference computation was not related to the length of A_k, the worst case response time of τ_k could be derived when $A_k = 0$. Based on this conclusion, the interference computation in our method dose not correspond to A_k. Thus, the worst case response time of τ_k can be derived in a similar way as [12] by the following theorem.

Theorem 1. *Let $\tilde{R}_k$ be the minimal solution of the (21) by operating an iterative fixed point search starting with $R_k = C_k$,*

$$R_k \leftarrow \left\lfloor \frac{\Psi(R_k)}{M} \right\rfloor + LC_k. \tag{21}$$

Then $\tilde{R}_k$ is an upper bound of τ_k's response time.

5. Evaluation

In this section, we evaluate both the accuracy and efficiency of our new analysis method with randomly generated task sets comparing with the technique proposed in [17].

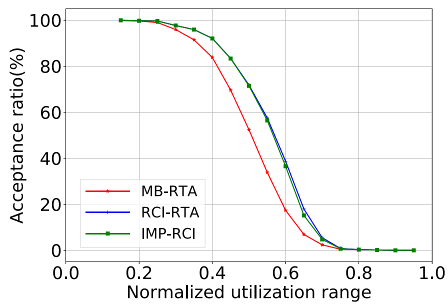
Task sets are generated in different strategies according to different experiments. But the range of each parameter for each sp-task is the same.

The number of segments s of each task is randomly generated from 1 to 5. If $s = 1$, T_i is uniformly generated in [100,1000] and C_i is uniformly generated in $[1, T_i/2]$. If $s > 1$, $m_{i,j}$ is uniformly generated in the interval $[1, M]$ and $LC_{i,j}$ is randomly chosen in the range $[1, T_i/(s \times M)]$ and T_i is uniformly generated in [100,1000].

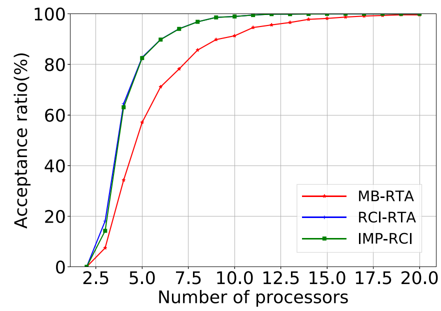
In the following, we use $MB-RTA$ to denote the method proposed in [17], $RCI-RTA$ to denote our new method proposed in Section 4.1 and $IMP-RCI$ to denote the improved efficiency method proposed in Section 4.1.2.

In the first setting of our experiments, we generate the task sets by the following strategy. First, a task set of $M+1$ tasks is generated. Then we iteratively increase the number of tasks with a step of one task to generate a new task set. This process is iterated until the normalized total utilization is larger than 1. The normalized task set utilization is defined as U_{total}/M. The whole procedure is repeated until a large sample space is generated. Based on these randomly generated task sets, we report the schedulability of tested methods as a function of the normalized task set utilization in [0,1]. The acceptance ratio obtained when $M = 4$ is showed in Fig. 7(a). The acceptance ratio of an analysis method is defined as the ratio between the number of task sets decided to be schedulable by this analysis method and the total number of tested task sets. When the utilization is small task sets are schedulable by three method and when the utilization is very large task sets are not schedulable by all methods. When the utilization of each task in the range [0.2,0.7], the curves show the difference. The trend of three curves shows that $RCI-RTA$ and $IMP-RCI$ clearly outperform $MB-RTA$ under all values of the system total utilization. And $IMP-RCI$ is not much worse than $RCI-RTA$. And the method $RCI-RTA$ improves the $MB-RTA$ at most 24% and the method $IMP-RTA$ improves the $MB-RTA$ at most 23%.

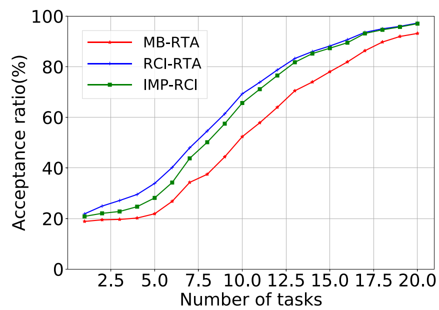
In the second setting, we generate the task sets using the same strategy but the constraint is that the total utilization of a generated task set cannot be larger than 2. we show the schedulability of all methods as a function of the number of processors M. The results for $U_{total} = 2$ when the number of processors is varied in [2,20] are presented in Fig. 7(b). From the results of the first experiment, when the normalized utilization is 0.5 the results show improvement clearly and the acceptance ratio is not very small and very large, so we use this utilization to do this experiment. For small values of M, our methods outperform $MB-RTA$. But



(a) $M = 4$

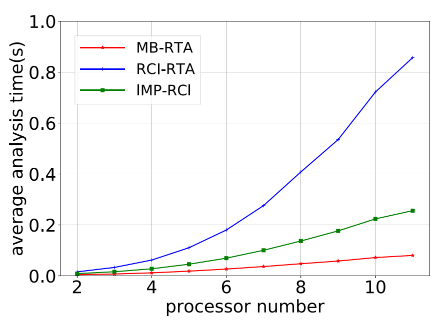


(b) $M \in (2, 20), U_{total} = 2$

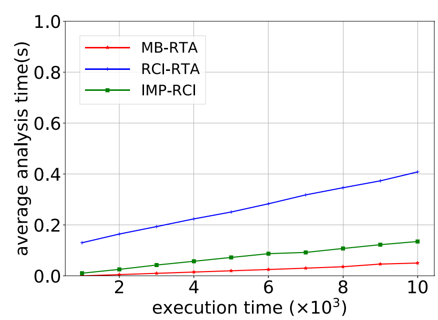


(c) $M = 4$, $U_{total} = 2.8$ and $n \in (2, 20)$

Fig. 7. Acceptance ratio of different parameters.



(a) With different number of processors



(b) With different range of execution time

Fig. 8. Evaluation of efficiency.

for a large number of M, all methods tend to a fully schedulable result. And the method $RCI - RTA$ improves the $MB - RTA$ at most 30% and the method $IMP - RTA$ improves the $MB - RTA$ at most 27%.

In the third experimental setting, we present the schedulability of the tested methods as a function of the number of tasks in one task set denoted by n. n is varied in [2,20], while $U_{total} = 2.8$ and $M = 4$. We use UUnifast [7] technique to generate task sets. The results are shown in Fig. 7(c). Because many light tasks (tasks with a small utilization) are easier to be scheduled than a few heavy tasks (tasks with a large utilization), the trend of the curves are from small to large. The curves of $RCI - RTA$ and $IMP - RCI$ clearly outperform $MB - RTA$ for any all value settings of number of tasks, even though both tests are converge to full schedulability for lager n. And the method $RCI - RTA$ improves the $MB - RTA$ at most 16% and the method $IMP - RTA$ improves the $MB - RTA$ at most 13%. Now, we evaluate the analysis efficiency of our new method. In Fig. 8, each curve denotes the average analysis time of one particular task set by a certain method. In Fig. 8(a), we set the range of T_i as [2, 1000]. If $s_i = 1$, the range of LC_i is set to [1, 1000]. Otherwise, we should generate each value of $LC_{i,j}$ and $LC_{i,j} \in [1, T_i/(s_i \times M)]$. The efficiency tests showed in the Fig. 8(a) is present as a function of the number of processors M range in [2,12]. In Fig. 8(b), we set the number of processors as 8, and change the upper limit of C_i. We can see that $RCI - RTA$ runs slower than $MB - RTA$. However, $IMP - RCI$ can significantly improve the analysis efficiency of $RCI - RTA$. $IMP - RCI$ method can handle complex task sets in acceptable time.

6. Conclusion

In this paper, we extend the state-of-the-art response time analysis method for sequential tasks to analyze sp-tasks. First, we achieve the worst case scenario of each sp-task in two cases depending on whether it has carry-in. Based on the worst case assumption, we propose a method to bound the interference of each sp-task, and then bound the total interference of all interfering tasks by considering the number of fore-carry-in threads. To further reduce the computation complexity, we propose an efficient method to compute the interference of each inter-task with carry-in. To derive an upper-bound of total interference of inter-tasks, we propose a dynamic programming algorithm. Finally, we conduct simulation tests to validate the performance of our methods in terms of both accuracy and efficiency with randomly generated sp-task sets.

Acknowledgment

This work is partially supported by NSF of China under grant nos. 61472072 and 61528202 and founded by Liaoning major equipment manufacturing collaborative innovation center.

References

[1] S. Baruah, Improved multiprocessor global schedulability analysis of sporadic DAG task systems, in: Proceedings of the Twenty-Sixth Euromicro Conference on Real-Time Systems, 2014, pp. 97–105, doi:10.1109/ECRTS.2014.22.
[2] S. Baruah, The federated scheduling of systems of mixed-criticality sporadic DAG tasks, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2016, pp. 227–236, doi:10.1109/RTSS.2016.030.
[3] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, The global EDF scheduling of systems of conditional sporadic DAG tasks, in: Proceedings of the Twenty-Seventh Euromicro Conference on Real-Time Systems, 2015, pp. 222–231, doi:10.1109/ECRTS.2015.27.
[4] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, A. Wiese, A generalized parallel task model for recurrent real-time processes, in: Proceedings of the Thirty-Third IEEE Real-Time Systems Symposium, 2012, pp. 63–72, doi:10.1109/RTSS.2012.59.
[5] S.K. Baruah, Techniques for multiprocessor global schedulability analysis, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2007, pp. 119–128.
[6] M. Bertogna, M. Cirinei, Response-time analysis for globally scheduled symmetric multiprocessor platforms, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2007.
[7] E. Bini, G.C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Syst. 30 (1) (2005) 129–154.
[8] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, A. Wiese, Feasibility analysis in the sporadic DAG task model, in: Proceedings of the Twenty-Fifth Euromicro Conference on Real-Time Systems, 2013, pp. 225–233, doi:10.1109/ECRTS.2013.32.
[9] A. Burns, S. Baruah, Sustainability in real-time scheduling, J. Comput. Sci. Eng. 2 (1) (2008) 74–97.
[10] H.S. Chwa, J. Lee, J. Lee, K.M. Phan, A. Easwaran, I. Shin, Global EDF schedulability analysis for parallel tasks on multi-core platforms, IEEE Trans. Parallel Distrib. Syst. 28 (5) (2017) 1331–1345, doi:10.1109/TPDS.2016.2614669.
[11] H.S. Chwa, J. Lee, K.M. Phan, A. Easwaran, I. Shin, Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms, in: Proceedings of the Twenty-Fifth Euromicro Conference on Real-Time Systems, 2013, pp. 25–34, doi:10.1109/ECRTS.2013.14.
[12] N. Guan, M. Stigge, W. Yi, G. Yu, New response time bounds for fixed priority multiprocessor scheduling, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2009, pp. 387–397.
[13] X. Jiang, X. Long, N. Guan, H. Wan, On the decomposition-based global EDF scheduling of parallel real-time tasks, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2016, pp. 237–246, doi:10.1109/RTSS.2016.031.
[14] S. Kato, Y. Ishikawa, Gang EDF scheduling of parallel task systems, in: Proceedings of the Thirtieth IEEE Real-Time Systems Symposium, 2009, pp. 459–468, doi:10.1109/RTSS.2009.42.
[15] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: Proceedings of the Thirty-First IEEE Real-Time Systems Symposium, 2010, pp. 259–268, doi:10.1109/RTSS.2010.42.
[16] J. Li, J.J. Chen, K. Agrawal, C. Lu, C. Gill, A. Saifullah, Analysis of federated and global scheduling for parallel real-time tasks, in: Proceedings of the Twenty-Sixth Euromicro Conference on Real-Time Systems, 2014, pp. 85–96, doi:10.1109/ECRTS.2014.23.
[17] C. Maia, M. Bertogna, L. Nogueira, L.M. Pinho, Response-time analysis of synchronous parallel tasks in multiprocessor systems, in: Proceedings of the Twenty-Second International Conference on Real-Time Networks and Systems, ACM, 2014, p. 3.
[18] C. Maia, L. Nogueira, L.M. Pinho, Scheduling parallel real-time tasks using a fixed-priority work-stealing algorithm on multiprocessors, in: Proceedings of the Eighth IEEE International Symposium Industrial Embedded Systems (SIES), 2013, pp. 89–92, doi:10.1109/SIES.2013.6601477.
[19] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, G. Buttazzo, Schedulability analysis of conditional parallel task graphs in multicore systems, IEEE Trans. Comput. 66 (2) (2017) 339–353, doi:10.1109/TC.2016.2584064.
[20] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, G.C. Buttazzo, Response-time analysis of conditional DAG tasks in multiprocessor systems, in: Proceedings of the Twenty-Seventh Euromicro Conference on Real-Time Systems, 2015, pp. 211–221, doi:10.1109/ECRTS.2015.26.
[21] A. Saifullah, K. Agrawal, C. Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, in: Proceedings of the Thirty-Second IEEE Real-Time Systems Symposium, 2011, pp. 217–226, doi:10.1109/RTSS.2011.27.

Meiling Han was born in Liaocheng, Shandong of China in 1988. She received her master's degree in computer applications technology from Shenyang Agricultural University, China in 2013, she is a Ph.D. candidate at the School of Computer Science and Engineering of Northeastern University, China. Her research interests are broadly in embedded real-time systems, especially the real-time scheduling on multiprocessor systems.

Tianyu Zhang received his master's degree in computer applications technology from the Northeastern University, China in 2013. Currently, he is a Ph.D. candidate at the School of Computer Science and Engineering at the Northeastern University in China. His research interests are broadly in the area of real-time embedded systems with a focus on real-time scheduling.

Qingxu Deng received his Ph.D. degree in computer science from Northeastern University, China, in 1997. He is a professor of the School of Computer Science and Engineering, Northeastern University, China, where he serves as the Director of institute of Cyber Physical Systems. His main research interests include Cyber-Physical systems, embedded systems, and real-time systems.