Journal of Circuits, Systems, and Computers
Vol. 27, No. 10 (2018) 1850165 (19 pages)

DOI: 10.1142/S0218126618501657

Global Fixed Priority Scheduling with Constructing Execution Dependency in Multiprocessor Real-Time Systems*

Meiling Han†, Tianyu Zhang‡, Yuhan Lin§, Zhiwei Feng¶
and Qingxu Deng‖

School of Computer Science and Engineering,
Northeastern University,
NO. 3-11 Wenhua Road Heping District,
Shenyang 110819, China
†*hanmeiling@stumail.neu.edu.cn*
‡*tyzhang@stumail.neu.edu.cn*
§*linyuhan@stumail.neu.edu.cn*
¶*fengzw@stumail.neu.edu.cn*
‖*dengqx@mail.neu.edu.cn*

Received 17 August 2017
Accepted 30 November 2017
Published 1 February 2018

The increasing demands for processor performance are driving system designers to adopt multiprocessors. In this paper, we study global fixed priority scheduling in multiprocessor real-time systems and introduce a technique for improving the schedulability. The key idea is to construct execution dependency for selected tasks to leverage slack time and reduce the interference between high-priority and low-priority tasks. Thus, more lower-priority tasks are enabled to be scheduled. Further, we provide a response time analysis method which takes the execution constraint of tasks into consideration. Extensive simulation results indicate that the proposed approach outperforms existing work in terms of acceptance ratio.

Keywords: Global fixed-priority scheduling; response time analysis; schedulability analysis; real-time systems.

1. Introduction

With the rapid development of multiprocessor systems, an increasing trend is to deploy real-time embedded systems on multiprocessor platforms. In such multiprocessor real-time systems, we need to guarantee the temporal correctness of tasks

*This paper was recommended by Regional Editor Tongquan Wei.
† Corresponding author.

running on parallel computing architectures. Thus, multiprocessor scheduling plays a critical role in multi-core system design to achieve desired computing performance.

Scheduling approaches in multiprocessor systems are usually classified into two categories: *partitioned* and *global*. In partitioned scheduling except semi-partition, the task set is partitioned into groups and each group is allocated to a single processor. Each task in the group is only allowed to be executed on the assigned processor even some other ones are currently idle. In a contrary, global scheduling allows tasks to be executed on any processor and even migrate from one to another at run-time. That is, automatic load balancing between processors is supported by global schedulers. Also, various mechanisms to reduce migration overhead are being proposed which further popularizes the adoption of global scheduling in multi-processors.[1,2] Some methods object to analyze tasks schedulability when they consider the thermal of multi-processors.[3,4] In this paper, we study global scheduling in multi-core real-time systems and focus on *fixed-priority* and *preemptive* scenarios, referred to as G-FP for the sake of simplicity.

The main difficulty on analysis of global fixed priority scheduling lies in the fact that the critical instant is in general unknown. Several attempts have been made by using either explicit or symbolic state space enumeration.[6,7] Geeraterts[8] improved the method proposed by Baker and Cirinei[6] by using an anti-chain technique. Sun *et al.*[9] proposed a similar method by using continuous time and linear hybrid automate. But all these methods run into serious scalability problems.

Another direction of tackling this challenge is to develop approximated methods in which sufficient conditions for schedulability test are desired.[5,10–12] Baker[13] proposed an analysis method based on the concept of problem window. The basic idea is to derive an upper bound on the interference contributed by each individual task, and then, the total interference of the system can be bounded. Bertogna *et al.*[14] further improved the analysis precision by giving a tighter bound on the interference of each task. The main contribution is the observation that if a high-utilization task has to execute in parallel with the analyzed task τ_k, the parallel part will not prevent the execution of τ_k. Based on this property, Bertogna and Cirinei[15] present the Response-Time Analysis (RTA) for global multiprocessor scheduling.

As another breakthrough, Baruah[16] firstly proposes to bound the number of carry-in tasks introduced in the 'problem window'-based analysis.[13] A carry-in task has one job which is released before the starting point of the problem window and finished within the problem window. A broad range of global scheduling algorithms, e.g.,[17–19] are proposed based on the carry-in bounding technique. In a most recent work,[19] Guan *et al.* propose the first worst-case RTA for tasks with arbitrary deadlines and the authors quantify the requested demand of higher-priority tasks by applying the workload function. We refer to the method in Ref. 19 as GSYY method throughout this paper.

A significant concept proposed in GSYY is the abstract critical instant which ends the unknown critical instant problem in global multiprocessor scheduling. Further more, Davis *et al.*[20] prove that the abstract critical instant is not a specific property only can be applied by GSYY, but a general property of G-FP scheduling itself. However, an observation from GSYY is that, in the worst case, the processors which are not assigned to the analyzed task τ_k may be idle while τ_k is executing. Such an execution pattern in which slack time exists can hurt the schedulability. Thus, in this work, we propose a technique to leverage the slack time and reduce the interference of the analyzed task from the higher-priority tasks. The basic idea is to increase the concurrent execution time between some higher-priority tasks and the analyzed task by constructing execution dependencies. After adding a task pair, we combine the RTA technique proposed in GSYY method to check all tasks schedulability. To analyze the schedulability of systems with execution dependencies, we present the schedulability test based on the RTA method in GSYY. To evaluate the proposed method, we conduct experiments with randomly generated task sets. Extensive results validate the improvement compared to GSYY in terms of system acceptance ratio.

The rest of the paper is organized as follows. In Sec. 2, we introduce the task model studied in this work. Section 3 gives a brief discussion on the RTA method of GSYY. In Sec. 4, we first give an motivational example and then present the mechanism that how to construct the execution dependency for selected tasks. Further, a corresponding RTA method is proposed in Sec. 5. Section 6 demonstrates the evaluation results and Sec. 7 concludes the paper.

2. System Model

This paper considers a sporadic task set running on a multiprocessor platform which consists of M identical processors. A task set τ is comprised by n tasks and each of them can release an infinite number of jobs. A task, denoted as τ_i, is represented by a 3-tuple of parameters: $\tau_i = (C_i, D_i, T_i)$, where C_i is the worst-case execution time (WCET), D_i is the relative deadline and T_i is the minimum interval between two successive released jobs of τ_i which is also called period, respectively.

Task systems can be classified into three categories according to the relationship between D_i and T_i of each task τ_i:

(i) *implicit-deadlines* system if $D_i = T_i, \forall \tau_i \in \tau$;
(ii) *constrained-deadlines* system if $D_i \leq T_i, \forall \tau_i \in \tau$;
(iii) *arbitrary-deadlines* system, otherwise.

In this paper, we consider a constrained-deadline system, i.e., $D_i \leq T_i$ for each task τ_i. At time t task τ_i is called *active* if it has a released job and this job is not finished at t. Thus, for any task τ_i in a constrained-deadline system, there is at most one job executed at any time.

The jth job of task τ_i is denoted as J_i^j. Its release time, finish time and absolute deadline are represented by r_i^j, f_i^j and $d_i^j = r_i^j + D_i$, respectively. The response time which equals to $f_i^j - r_i^j$ is denoted as R_i^j. We use J_i to denote a job of task τ_i when it is clear from the context which job of τ_i we are referring to, and use r_i, f_i and d_i to denote its release time, finish time and absolute deadline, respectively. The exact worst case response time (WCRT) of τ_i is defined as the maximum response time among all its jobs. Naturally, the exact WCRT can be achieved by enumerating all jobs' response time. However, such a manner is not practical for an infinite sequence of jobs. To overcome this, a common method is using an approximate technique to get an upper bound of the response time for operating schedulability test for a task set. We use R_i to denote the achieved response time upper bound for τ_i, i.e., $R_i \geq \max R_i^j, \forall j \in \{1, 2, \ldots, N^+\}$. Therefore, a task set is called schedulable if the response time is less than or equal to the relative deadline for each task, i.e., $R_i \leq D_i$, $\forall \tau_i \in \tau$. We use $U_i = \frac{C_i}{T_i}$ to denote the utilization of task τ_i. $U = \sum_{i=1}^{N} U_i$ is the total utilization of the task set.

We assume the system is fully pre-emptive and adopts a Global Fixed Priority (G-FP) scheduling mechanism. Specifically, each task is assigned with a unique priority level and all tasks are sorted according to their priorities. We define that τ_i has a higher priority than τ_j if $i < j$. At any time point, as long as an idle processor exists, the job of the most highest priority task in the ready queue is allowed to be executed.

For simplicity of expression, we use the following notations to express that a value A is "limited" if it is bounded by a threshold value. $[\![A]\!]_B = \max(A, B)$, $[\![A]\!]^C = \min(A, C)$, 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 $A > C$.

Finally, we assume that the system is modelled using discrete time, and use $[t, t+1)$ to denote a time interval of unit length. All events in the system happen at integer clock ticks $1, 2, \ldots$.

3. GSYY Method

In this section, we give a brief review of the GSYY method proposed by Guan *et al.*[19] in which a RTA for G-FP scheduling is presented. The main objective of RTA is to analyze and achieve an upper bound of the response time for each task. In the following, we refer to the analyzed task as τ_k and J_k is the job of τ_k having the maximum response time.

GSYY method assumes that the busy period of J_k starts at r_k (the release time of the analyzed job J_k), and at least one of the following conditions holds at any time point during the busy period:

(i) All processors are busy.
(ii) J_k is executing.

J_k finishes the execution at the end of the busy period. Then, the objective of GSYY method is to analyze the minimum length of the busy period which ensures J_k can be finished.

GSYY method computes an upper bound on the maximum interference of each individual task in the busy period, and uses the sum of them as a safe upper bound of the total interference of all higher-priority tasks. The interference of a higher-priority task τ_i can be achieved in two steps. First, the maximum workload of τ_i that may execute in the busy period is computed. Then, τ_i's interference is the part of its workload that can actually prevent the analyzed task τ_k from executing.

The workload of a task in J_k's busy period can be divided into three parts (see Fig. 1):

(i) *carry-in*: the contribution of at most one job (called carry-in job) with release time earlier than the busy period and deadline in the busy period;
(ii) *body*: the contribution of all jobs (called body jobs) with both release time and deadline in the busy period;
(iii) *carry-out*: the contribution of at most one job (called carry-out job) with release time in the busy period and deadline after the busy period.

The worst case of J_k has been proved to be appeared at when there are at most $M - 1$ higher priority tasks executing at $r_k - 1$ and the rest higher priority tasks are released at r_k. This can be summarized as the following theorem.

Theorem 1. *There are at most $M - 1$ tasks having carry-in, and for each task τ_i, the carry-in is at most $C_i - 1$.*

Proof. Refer to GSYY method.[19] □

Then, the workload of a task $\tau_{i<k}$ can be computed by considering two cases, i.e., whether carry-in exists. The total workload in J_k's busy period is bounded by Bertogna *et al.*[14,15] They find out that if the workload of τ_i is too large, then some workload would be executed parallel with J_k, and would not prevent J_k from executing. So the actual workload which interferes with J_k is upper bounded by $L - C_k + 1$. Note that the upper bound of τ_i's interference is $L - C_k + 1$ rather than $L - C_k$, to facilitate the iterative RTA procedure. A formal explanation of this issue can be found in Ref. 14.

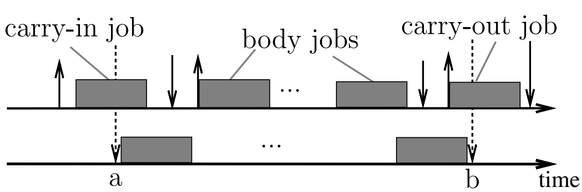


Fig. 1. Three execution parts of a task in a busy interval.

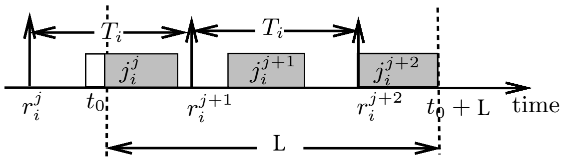


Fig. 2. The release pattern of a task if carry-in job exists.

If task τ_i has carry-in, we use $W_i^{\mathrm{CI}}(\tau_i, L)$ to denote the worst case workload that τ_i generates in a busy period with length L as shown in Fig. 2. $W_i^{\mathrm{CI}}(\tau_i, L)$ can be computed by Eq. (1).

$$W^{\mathrm{CI}}(\tau_i, L) = \left(\left\lfloor \frac{(L - C_i)}{T_i} \right\rfloor + 1 \right) \times C_i + \alpha\,, \tag{1}$$

where $\alpha = [\![[\![[\![(L - C_i)]\!]_0 \bmod T_i - (T_i - R_i)]\!]_0]\!]^{C_i}$. Then, the interference of task τ_i to τ_k in a busy period with length L is computed by Eq. (2).

$$I^{\mathrm{CI}}(\tau_i, L) = [\![W^{\mathrm{CI}}(\tau_i, L)]\!]^{L - C_k + 1}\,. \tag{2}$$

If task τ_i does not have carry-in, we use $W_i^{\mathrm{NC}}(\tau_i, L)$ to denote the worst case workload that task τ_i generates in a busy period with length L.

$$W^{\mathrm{NC}}(\tau_i, L) = \left(\left\lfloor \frac{L}{T_i} \right\rfloor \right) \times C_i + \beta\,, \tag{3}$$

where $\beta = [\![L \bmod T_i]\!]^{C_i}$. Then, the interference of task τ_i in a busy period with length L is computed by Eq. (4).

$$I^{\mathrm{NC}}(\tau_i, L) = [\![W^{\mathrm{NC}}(\tau_i, L)]\!]^{L - C_k + 1}\,. \tag{4}$$

A linear method is proposed to get the largest $M - 1$ carry-in interference, by considering the difference of interference between cases of having carry-in or not. The difference is called $Idiff$ and defined as follows:

Definition 1. *Idiff* of task τ_i is denoted by *Idiff*$_i$ and defined as:

$$Idiff(\tau_i, L) = I_i^{\mathrm{CI}}(\tau_i, L) - I_i^{\mathrm{NC}}(\tau_i, L)\,. \tag{5}$$

Then, the total interference of higher priority tasks is defined as the sum of the largest $M - 1$ $Idiff$s and all interference when higher priority tasks have not carry-in, see Eq. (6).

$$\Omega_k(x) = \left(\sum_{\tau_i \in \tau} I^{\mathrm{NC}}(\tau_i, x) + \sum_{\text{the } (M-1) \text{ largest}} Idiff(\tau_i, x) \right). \tag{6}$$

Since the busy period of τ_k ends when it finishes its execution, the length of the busy period indicates the response time of τ_k which is upper bounded by the following theorem in GSYY method.[19]

Theorem 2. *Let χ be the minimal solution of the following equation by doing an iterative fixed point search of the right-hand side staring with $x = C_k$.*

$$x = \left\lfloor \frac{\Omega_k(x)}{M} \right\rfloor + C_k\,. \tag{7}$$

Then, χ is a safe upper bound for τ_k's response time.

4. Motivation and Rules

In this section, we give the motivation about adding dependency among higher priority tasks when there are tasks that cannot be schedulable. Then, we define the dependency relationship between two tasks as a task pair and present definitions in the presented task pair model. Finally, we discuss the rules that how to select tasks to construct a task pair.

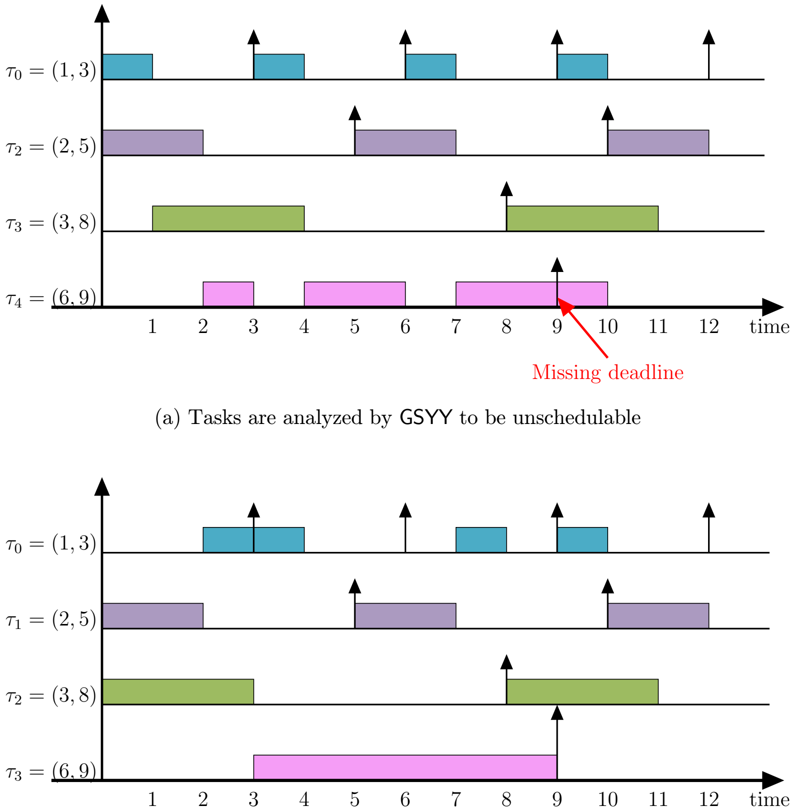
4.1. *Motivation and definitions*

From the discussion of GSYY method, an observation is that, when the analyzed task is executing, the rest processors may be idle. Also, some higher priority tasks may have completed their executions far early before their deadlines meanwhile a task with lower priority may miss its deadline. If a higher priority task executes later by bounding its execution with another higher priority task's execution, the idle time interval paralleled with the analyzed task can be reduced. Then, the analyzed task may execute earlier and finish before its deadline. Thus, the schedulability of the task set can be improved as shown in Example 1.

Example 1. Consider a task set consists of four tasks and each of them is denoted as $\tau_i = [C_i, T_i](T_i = D_i)$. Specifically, $\tau_0 = [1,2], \tau_1 = [2,5], \tau_2 = [2,7], \tau_3 = [5,8]$. They are scheduled on a multi-processor platform with 2 processors.

Figure 3(a) shows the worst-case of the task set analyzed under GSYY method and task τ_3 misses its deadline at 8. If we select τ_0 and τ_1 to construct a task pair, τ_0 cannot be active when τ_1 is active. τ_0, τ_1 and τ_2 release their first job at 0, τ_0 is not active until time slot 2 when τ_1 finishes its execution. At time 0, τ_0 cannot be executed. Then, τ_2 can be executed which is earlier comparing to GSYY method. Finally, τ_3 finishes its execution at 8 and all tasks are schedulable as shown in Fig. 3(b).

In Example 1, a task with lower priority like τ_2 whose execution is effected by higher tasks like τ_0. If these two tasks are chosen as a task pair, then the execution



(a) Tasks are analyzed by GSYY to be unschedulable

(b) τ_0 and τ_2 are not allowed to execute at a same time

Fig. 3. Examples for our scheduling method intuition.

sequence of two tasks are changed. The intuition of this paper is that making some higher priority tasks executing later, then tasks with lower priority may allowed to be executing earlier.

From Example 1, we observe that the worst case response time of τ_0 is increasing, while the worst case response time of τ_2 and τ_3 are decreasing. Because the idle time interval paralleled with τ_2 and τ_3 decreases and are used by τ_0. Such an observation motivates us to propose a concept called task pair to leverage the idle time paralleled with the analyzed task.

We assume that there are p task pairs in a task set, and there is a dependency relationship between tasks in a task pair. We also assume that a task can only be allowed in one task pair, that is, there are at most $\lceil n/2 \rceil$ task pairs in the system. A task pair (τ_i, τ_j) consists of τ_i and τ_j is denoted as $Pa_i = Pa_j = (i, j)$, and τ_p denotes the set of all task pairs. Then, the dependency relationship between tasks τ_i and τ_j is defined by the following definition.

Definition 2. If task τ_i and task τ_j compose $Pa_{i,j}$ and $i < j$, then at any time instant, if τ_j is active then τ_i cannot be active until f_j, where f_j is the finish time of a job of τ_j.

According to the relationship between two tasks in $Pa_i = (i, j)$, when a job of τ_j is released then it can block task τ_z from execution, where $i \leq z < j$. Note that, when τ_j is active, any job of τ_i cannot be executed on any processor even though there are idle processors. Then, all the tasks between τ_i and τ_j can block the execution of τ_i. So the total interference that can interfere τ_i from executing can be divided into two types.

Definition 3. The higher priority tasks interference on task τ_i is defined as the cumulative time that higher priority tasks prevent task τ_i from executing.

Definition 4. The lower priority tasks interference on task τ_i is defined as the cumulative time that lower priority tasks prevent task τ_i from executing.

Now the problem is how to select task pairs in the higher priority sub-task set. When a task has carry-in in the busy period, we find that if the worst case response time of any task is increasing, then the length of carry-in job of it is also increasing. And the final result is increasing the interference to lower priority tasks. In the contrast, the worst case response time of any task is decreasing, then the length of carry-in job of it is decreasing too. Thus, the interference to lower priority tasks is decreased. According to the observation, we propose a method to select task pairs.

4.2. *Selecting task pairs*

According to the discussion above, we discuss how to select task pairs in this section. We assume τ_k is the first task missing its deadline when analyzing by GSYY method. Suppose the tasks with higher priorities are in τ_{hp}, we select one task pair in τ_{hp} according to some rules. If τ_k is schedulable, we stop selecting task pairs. Otherwise, the selecting continues until τ_k is schedulable or there are no tasks to select. We assume there are p task pairs existing in τ_{hp}.

To avoid adding too much complication to analysis of tasks, we assume that one task is only allowed in one task pair.

Rule 1. τ_i is allowed to be selected in Pa_i if $i \notin \forall pa_z \wedge \tau_z \in \tau_p$.

The higher priority task in a task pair would get more interference which results a larger response time. Our purpose is to improve the schedulability of a task set, so the increasing response time of tasks in τ_p cannot be larger than their deadlines. That is, we should ensure that, τ_{hp} is still schedulable after selecting task pairs. This means that if $Pa_i = (i, j), i < j$, τ_i should finish its execution before its deadline even in the worst case. This observation is summarized as the second rule.

Algorithm 1. Selecting task pairs in τ_{hp}

Require: p, τ_{hp}, M
Ensure: All tasks in τ_{hp} are schedulable

function ChooseTPair(p, τ_{hp}, M)
  $RT =$ GSYY $(Tset, M)$
  $Lt\_set = late(\tau_{hp}, RT_{hp})$
  while $P \geq 0$ **do**
    $i = max\_index(Lt\_set)$
    $j = i + 1$
    while $j - i \leq len(\tau_{hp} - j)$ **do**:
      if all tasks in $\tau_h p$ are schedulable **then**
        $tp = (i, j)$
        $\tau_{hp}.delete(\tau_i, \tau_j)$
        break
      else
        $j = j + 1$
        $tp = []$
      end if
    end while
    if $tp! = NULL$ **then**
      $Tpair.add(tp)$
      $p = p - 1$
    end if
    if $p > len(\tau_{hp})/2$ **then**
      break
    else
      $Lt\_set.delete(max(Latency))$
    end if
  end while
  if $Len(Tpair)! = p$ **then**:
    $Tpair = NULL$
  end if
  return $Tpair$
end function

Rule 2. τ_i and τ_j are allowed to be selected in a task pair if all tasks in $\{\tau_z | \tau_z \in \tau_{hp} \cap \min(i, j) < z\}$ are schedulable.

Rule 2 comes from that the increasing response time of $\tau_i, i < j$ can only effect the tasks with lower priorities than τ_i in τ_{hp}, and this may increase the interference for these lower priority tasks. Our objective is to make the task set be schedulable, so the schedulability of tasks in τ_{hp} can still be guaranteed.

When there only one task pair $Pa_i = (i, j)$ exists, we consider the changing trend of the WCRT of task τ_z, where $i < z < j$. The WCRT of τ_z may have been increased, the reason will be discussed in Sec. 5. Since the WCRT of some higher priority tasks are increased, the interference of them to lower priority tasks must increase according to Eq. (1). We should ensure that after adding a task pair, the WCRT of tasks with priorities lower than τ_j would be reduced, which means the increasing interference of tasks between τ_i and τ_j cannot be larger than the reducing interference of τ_i and τ_j. That is, the lower priority tasks have opportunities to execute early. According to the discussion above, a simple rule is that the number of tasks between τ_i and τ_j is upper bounded by the number of tasks with priorities lower than τ_j, meanwhile the WCRT of tasks with lower priorities of lower than τ_j cannot increase.

Based on the discussion above, we can select a task pair from τ_{hp} for the analyzed task. For a task set, we use GSYY method to get the first task τ_k missing deadline, and all higher priority tasks in τ_{hp} and their response time in RT. For any task $\tau_i \in \tau_{hp}$, we compute its latency $L_i = D_i - R_i$, and $Lt\_set[i] = L_i$ which denotes the set of all higher priority tasks latency. To select a task pair, we first choose τ_i with the largest latency of Lt_set as the higher priority task in a task pair. Then, we select the lower priority task of this task pair through the rules described above. If no such task exists, then we delete L_i in Lt_set, and continue to select τ_i with largest latency in Lt_set. After constructing a task pair, these tasks cannot be selected again and we delete them from τ_{hp}. This procedure is repeated until we get p task pairs. We give the details of selecting p task pairs in Algorithm 1.

A task with larger latency can be delayed with longer time, then the lower priority tasks can get more opportunities to finish their executions before their deadlines. Therefore, for each task pair, we prior select the task with largest latency as the higher priority task in a task pair and then select a task with lower priority. The optimal result can be achieved by enumerating all possible combinations of task pairs, which runs into a scalability problem. And our method can get a feasible results in an acceptable complexity. Next, we show how to do interference analysis of a task set with task pairs.

5. RTA with Task Pairs

In this section, we discuss how to analyze the schedulability of a task set with task pairs. We assume that task τ_k is the first task missing its deadline, and p task pairs are already constructed to improve τ_k's response time. According to the discussion above, all tasks that can prevent τ_k from executing can be divided into the following parts:

(i) $Hp(\tau_k) = \{\tau_i \mid \tau_i \in \tau \wedge i < \min(Pa_k)\ \&\ i \notin \forall pa_z \wedge pa_z \in \tau_p\}$
(ii) $Hp(\tau_k)_p = \{\tau_i \mid i \in pa_i \wedge pa_i \in \tau_p\ \&\ i < \min(Pa_k)\ \&\ i \notin \forall pa_z \wedge \tau_z \in Hp(\tau_k)_p\}$
(iii) $Lp(\tau_k) = \{\tau_i \mid k \leq i \leq j\ \&\ Pa_j = Pa_k \wedge k < j\ \&\ i \notin \forall pa_z \wedge pa_z \in \tau_p\}$
(iv) $Lp(\tau_k)_p = \{\tau_i | k \leq i \leq j\ \&\ Pa_j = Pa_k \wedge k< j\ \&\ i \in pa_i \wedge pa_i \in \tau_p\ \&\ i \notin \forall pa_z \wedge \tau_z \in Lp(\tau_k)_p\}\}$
(v) $\tau_j, Pa_j = Pa_k\ \&\ k < j$

Obviously, $Hp(\tau_k) \cup Hp(\tau_k)_p$ is the set of tasks with higher priorities than τ_k, and $Lp(\tau_k) \cup Lp(\tau_k)_p$ is the set of tasks with lower priorities which can block τ_k from executing. Each task that prevent τ_k from executing just is counted once, then we have $Hp(\tau_k) \cap Hp(\tau_k)_p = \emptyset$ and $Lp(\tau_k) \cap Lp(\tau_k)_p = \emptyset$. For all tasks which can prevent τ_k form executing are departed into two case depending on whether it is in a task pair. To avoid redundant analysis of tasks in a task pair, we take them as a union. If $\tau_{i \wedge i<k}$ not in any task pair, then it is just a normal higher priority task to prevent τ_k form executing. If $\tau_{i \wedge i<k}$ is in a task pair, then the interference of τ_i is discussed as task pair pa_i's interference together with $\tau_h \in pa_i$. Since, $Hp(\tau_k)_p$ only has the higher priority task in a task pair. We discuss the reason about lower priority tasks partition by considering whether τ_k is in a task pair. If τ_k is not in a task pair, i.e., $Pa_k = \emptyset$, then $\min(Pa_k) = k$ and $Lp(\tau_k) \cup Lp(\tau_k)_p = \emptyset$. If τ_k is in a task pair and τ_k with a higher priority than the other task in a task pair. Then, the tasks with priority between these two tasks can block τ_k from execution, due to the definition of task pairs. If task τ_i is a such task and not in any task pair, then it is as a normal lower priority task to block τ_k. Else if τ_i is a such task and in a task pair, then the interference of τ_i is discussed together with $\tau_l \in Pa_i$ as a task pair's interference. Since, just considering the task with higher priority of each task pair in $Lp(\tau_k)_p$ is enough.

5.1. *Interference from higher priority tasks*

For τ_k, there are two classes of higher priority tasks depending on whether they are in any task pair. For tasks in $Hp(\tau_k)$, the interference can be classified into two types depending on whether carry-in exists. If $\tau_i \in Hp(\tau_k)$, its interference can be computed by Eq. (2) if it is with carry-in and by Eq. (4) if it is without carry-in. And the $Idiff$ of τ_i denoted by $Idiff(\tau_i, L)$ can be computed by Eq. (5). If $\tau_i \in Hp(\tau_k)_p$, we denote the interference if Pa_i has carry-in as $I_p^{\mathrm{CI}}(Pa_i, \tau_i, L)$, and if Pa_i does not have carry-in as $I_p^{\mathrm{NC}}(Pa_i, \tau_i, L)$.

A. If $Pa_j = Pa_i,\ j = Pa_i \backslash i$ **and** $i < j < k$

In this case, τ_j and τ_i both interfere with τ_k & $j = Pa_i \backslash i$. Due to the behavior of τ_j and τ_i, at most one of them is allowed to have carry-in. For the sake of safety, we select the one with largest $Idiff$ to have carry-in.

If Pa_i has carry-in, we discuss the worst case situation. In this case, we just care about the worst case of workload they can generate in a busy period with length L. If τ_i has carry-in, τ_j cannot have carry-in. Then, the worst case workload that τ_i generates can be computed by Eq. (1), and the worst case workload that τ_j generates can be computed by Eq. (3). In the busy period of τ_k, τ_i and τ_j cannot execute in parallel with each other, then the actual interference to τ_k can be computed by the following equation:

$$I_p^{i\mathrm{CI}}(Pa_i, \tau_i, L) = [\![W^{\mathrm{CI}}(\tau_i, L) + W^{\mathrm{NC}}(\tau_j, L)]\!]^{L-C_k+1} . \tag{8}$$

Then, if τ_i does not have carry-in and τ_j has carry-in, the worst case interference can be computed as following:

$$I_p^{i\mathrm{NC}}(Pa_i, \tau_i, L) = [\![W^{\mathrm{NC}}(\tau_i, L) + W^{\mathrm{CI}}(\tau_j, L)]\!]^{L-C_k+1}. \tag{9}$$

Then, the worst case interference when Pa_i has carry-in equals the maximum interference of this two cases, see Eq. (10).

$$I_p^{\mathrm{CI}}(Pa_i, \tau_i, L) = [\![I_p^{i\mathrm{CI}}(Pa_i, \tau_i, L)]\!]_{I_p^{i\mathrm{NC}}(Pa_i, \tau_j, L)} . \tag{10}$$

If Pa_i does not have carry-in, then the worst case interference can be computed by Eq. (11).

$$I_p^{\mathrm{NC}}(Pa_i, \tau_i, L) = [\![W^{\mathrm{NC}}(\tau_i, L) + W^{\mathrm{NC}}(\tau_j, L)]\!]^{L-C_k+1} . \tag{11}$$

Then, the $Idiff$ of Pa_i is denoted as $Idiff_p(Pa_i, \tau_i, L)$, and can be computed by Eq. (12) when considering τ_i.

$$Idiff_p(Pa_i, \tau_i, L) = I_p^{\mathrm{CI}}(Pa_i, \tau_i, L) - I_p^{\mathrm{NC}}(Pa_i, \tau_i, L). \tag{12}$$

B. If $Pa_j = Pa_i,\ j = Pa_i \backslash i$ and $i < k < j$

Notice that we just consider the worst case interference. And in τ_k's busy period, the situation of pa_i's execution can be divided into three cases. The first case happens when τ_j is released before r_k and finishes in parallel with τ_k's execution, which means there is no chance for τ_i to execute. The interference in this case is denoted by $I_{j,k}(\tau_j)$. The second case happens when a job of τ_j has finished its execution before r_k and the next job of τ_j is not released. Then, the first job of τ_i is executed in its worst case and the rest jobs released in this busy period are with the original priority which can interfere τ_k, and its interference is denoted by $I_{i,k}(\tau_i)$. The third case happens when τ_j finishes in the busy period of τ_k and its next job is not released. Then, the first job of τ_i with priority of τ_j can not prevent τ_k from executing. If there is no new released job of τ_j, the new released job of τ_i returns to use its original priority and interfere τ_k, and the interference is denoted by $I_k(Pa_i)$. The extreme value of $I_k(Pa_i)$ is found in the first two cases. Because, only tasks with priority i can interfere τ_k. Obviously, the following inequation holds:

$$\min(I_{i,k}(\tau_i), I_{j,k} \leq (\tau_j))I_k(Pa_i) \leq \max\left(I_{i,k}(\tau_i), I_{j,k}\right).$$

For the case, there is only τ_i executing in τ_k's busy period, τ_i is as a normal higher priority task contributing worst case interference (computed by Eqs. (2) and (4)), with an increasing WCRT. For the case, there is only τ_j executing in τ_k's busy period, τ_j executes with the priority of τ_i. Thereby, τ_j can prevent τ_k from executing and the interference of τ_j can be computed by Eqs. (2) and (4). So we select the case that generates the largest $Idiff$ if Pa_i has carry-in. If Pa_i does not have carry-in, then we select the task with the largest interference in pa_i.

Lemma 1. *If $Pa_j = Pa_i$, $i < k < j$, and Pa_i has carry-in, the worst case $Idiff$ is*:

$$Idiff_p(Pa_i, \tau_i, L) = [\![Idiff_i(\tau_i, L)]\!]_{Idiff_i(\tau_j, L)}$$

Proof. According to the discussion above, the lemma holds obviously. □

We refer to the task with the maximum $Idiff$ as τ_z. Then, we have $Idiff(\tau_z, L) = Idiff_p(Pa_i, \tau_i, L)$, and the interference of Pa_i without carry-in is computed by Eq. (13).

$$I_p^{\mathrm{NC}}(Pa_i, L) = I^{\mathrm{NC}}(\tau_z, L)\,. \tag{13}$$

If $Idiff_p(Pa_i, \tau_i, L)$ does not belong to the largest $M-1$ $Idiff$ set, the worst case interference of $Pair_i$ is computed as follows:

$$I_p^{\mathrm{NC}}(Pa_i, L) = [\![I^{\mathrm{NC}}(\tau_i, L)]\!]_{I^{\mathrm{NC}}(\tau_j, L)}\,. \tag{14}$$

Currently, we have stated how to analyze the interference of higher priority tasks under all different cases. Next, we study the interference of lower priority tasks.

5.2. *Interference from lower priority tasks*

If τ_k does not belong to any task pair or $k \in Pa_k$ & $k > (Pa_k \backslash k)$, then $Lp(\tau_k) \cup Lp(\tau_k)_p = \emptyset$. If $k \in Pa_k$, we assume that $j \in Pa_k$ & $j > k$. According to the definition of task pair, τ_k is executed with the priority of τ_j, and τ_j is executed with the priority of τ_k. Then, the tasks having priorities lower than τ_k and higher than τ_j can block the execution of τ_k. If $\tau_i \in Lp(\tau_k)$, the analysis is similar with the analysis of higher priority tasks in $Hp(\tau_k)$. And if $\tau_i \in Lp(\tau_k)_p$, the analysis is similar with the analysis of higher priority tasks in $Hp(\tau_k)_p$ as discussed in Sec. 5.1.

According to the definition of tasks pair, there is a special task which can prevent task τ_k form executing when $k \in Pa_k$ & $k < (Pa_k \backslash k)$. We refer to this task as τ_j. If τ_j is active, τ_k cannot be executed. So τ_j can interfere τ_k from executing, which means τ_j can not be executed in parallel with τ_k. τ_k is not active until τ_j finishes its execution, even though idle processors exist. Since the worst case workload of a task during a time interval appears when it has carry-in, the interference of τ_j is the workload of its carry-in case. Finally, a lower bound of the total interference of all interfering tasks can be achieved by the following lemma.

Lemma 2. *The total interference $\Omega_k(L)$ of all interfering tasks must satisfy the following conditions*:

$$\Omega_k(L) \geq W^{\mathrm{CI}}(\tau_j, L) \times M\,.$$

Proof sketch. According to Definition 1, τ_k cannot be executed until τ_j finishes its execution. Therefore, the lemma holds. □

5.3. *Procedure of RTA*

So far, we have presented all cases of interfering tasks. According to Theorem 2, we compute the total interference by there steps. First, compute the sum of the largest $M-1$ $Idiff$s, and the sum is denoted as $Idiff_{\mathrm{sum}}$. Second, compute the total interference of all interfering tasks when they do not have carry-in. In this step, tasks are departed as it is with higher priority or lower priority. We define the total interference of all tasks with higher priorities in Eq. (15) and the total interference of all tasks with lower priority in Eq. (16). Finally, the total interference of all interfering tasks is defined by Eq. (17).

$$\Omega_k^{Hp}(L) = \sum_{i \in Hp(\tau_k)} I^{\mathrm{NC}}(\tau_i, L) + \sum_{i \in Hp(\tau_k)_p} I_p^{\mathrm{NC}}(Pa_i, \tau_i, L)\,, \tag{15}$$

$$\Omega_k^{Lp}(L) = \sum_{\tau_i \in Lp(\tau_k)} I^{\mathrm{NC}}(\tau_i, L) + \sum_{i \in Lp(\tau_k)_p} I_p^{\mathrm{NC}}(Pa_i, \tau_i, L)\,, \tag{16}$$

$$\Omega_k(L)^j = Idiff_{\mathrm{sum}} + \Omega_k^{Hp}(L) + \Omega_k^{Lp}(L)\,. \tag{17}$$

We observe that $\Omega_k(L)^j$ does not consider the interference from τ_j, where $j \in Pa_k$. According to Lemma 5.2, the total interference of all interfering tasks including τ_j is defined as follows:

$$\Omega_k(L) = [\![\Omega_k(L)^j - W^{\mathrm{CI}}(\tau_j, L) \times (M-1)]\!]_0 + W^{\mathrm{CI}}(\tau_j, L) \times M\,. \tag{18}$$

According to Theorem 2 and the discussion above, we finally get the response time of τ_k according to the following theorem.

Theorem 3. *Let χ be the minimum solution of the following Eq. (19) by doing an iterative fixed point search of the right side staring with $x = C_k$.*

$$x = \left\lfloor \frac{\Omega_k(L)}{M} \right\rfloor + C_k\,. \tag{19}$$

Then, χ is a response time upper bound for τ_k.

Proof: The proof of the theorem can be established in a similar way as GSYY method. □

6. Evaluation

We evaluate both the precision and efficiency of our new analysis with randomly generated task sets, comparing with GSYY method.

Task sets are randomly generated by the following strategy. First, a task set of $M + 1$ tasks is generated. Then, we increase the number of tasks as a step of 1 to construct a new task set. This process repeats until total utilization of the task set is larger than the utilization that you asked. The whole procedure is then repeated, starting with a new task set of $M + 1$ tasks, until a reasonably large sample space is generated. To generate each task, T_i is randomly picked from a union of $[2, 100]$, and C_i is randomly chosen in $[1, T_i/2]$. All tasks have implicit deadlines, i.e., $D_i = T_i$.

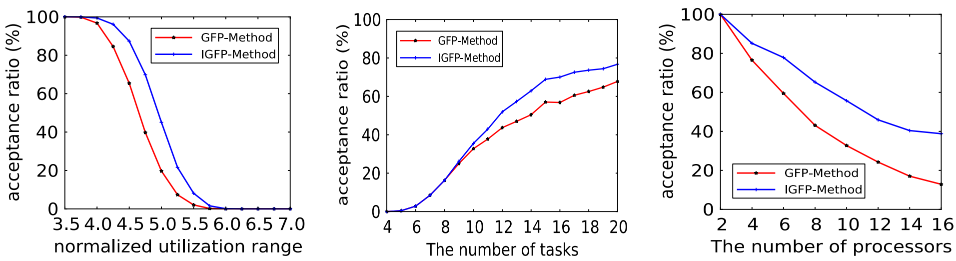
6.1. *Precision*

We conduct three simulation tests to evaluate the performance of our method by comparing with GSYY method. All the tests show the evaluation results in terms of acceptance ratio with different varying values. The acceptance ratio is defined as the ratio between the number of schedulable task sets and the total number of tested task sets. GFP_Method and IGFP_Method denote GSYY and our proposed method in this paper, respectively. We used rate monotonic scheme to assign tasks priorities.

The first test shows the acceptance ratio curves of two methods when $M = 8$, in Fig. 4(a). In this test, the system utilization U was varied in $[3.5, 7.0]$. The curves show that IGFP_Method and GFP_Method are indistinguishable and IGFP_Method is over performed than GFP_Method when the system utilization U was varied in $[4.0, 6.0]$.

In the following tests, We used UUnifast[21] to derive individual task utilization for a fixed value of n when we already know the system utilization. We generated individual task period randomly in $[2, 100]$. Finally, we used $U_i \times T_i$ to get τ_i's WCET. Then, we sorted the tasks in a task set by their priorities.

The second test reports how IGFP_Method performed when the number of tasks n is varied from 4 to 20 by step of 1, while $M = 8$ and the system utilization is 50%, in Fig. 4(b). IGFP_Method outperforms GFP_Method when n is larger than 10,



(a) Varying U (b) Varying n (c) Varying M

Fig. 4. Acceptance ratio with varying U, n, M.

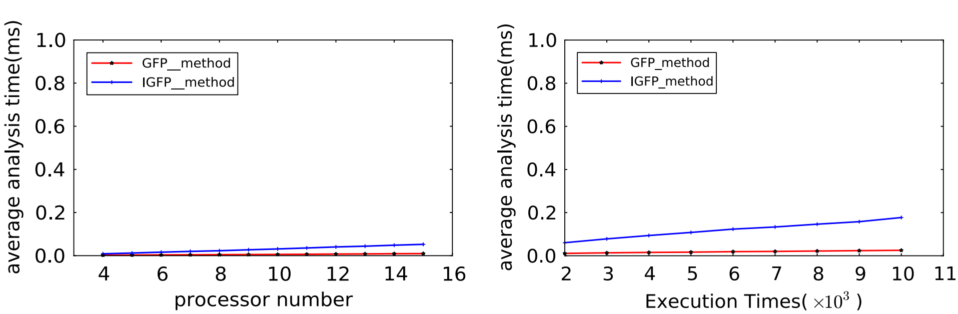
although both methods tend to full schedulability for larger n. Intuitively, many light tasks are much easier scheduled than few heavy tasks.

The final test illustrates the schedulability as function of number of processors which is varied according to the sequence $[2, 4, 6, 8, 10, 12, 14, 16]$, with $U = 0.5 \times M$ and $n = 1.5 \times M$, in Fig. 4(c). The schedulability of both methods degrades for high values of M, but IGFP_Method degrades much slower than GFP_Method, while the first one outperforms the last one.

6.2. *Efficiency*

We also evaluate the analysis efficiency of our new method. In Fig. 5, the two curves are: $GFP - Method$ denotes the average time it takes GSYY method to analyze one task set and $IGFP - Method$ denotes the average analysis time of one task set of our new method.

In Fig. 5(a), we set the rage of T_i as $[2, 2000]$, thus the range of C_i is $[1, 1000]$. Then, we vary the number of processors M. In Fig. 5(b), we set the number of processors as $M = 10$, and vary the upper bound of C_i. Through this two tests, we



(a) Varying M (b) Varying C_i

Fig. 5. Evaluation of efficiency with varying M and C_i.

can see our method is slower than GSYY method. However, our methods also can handle complex task sets in very short time.

7. Conclusion

In global fixed-priority scheduling, a lower priority task can miss its deadline meanwhile some higher priority tasks finish executing far earlier than their deadlines. If the higher priority tasks execute later, there are chances for lower priority tasks to execute. In this paper, we construct a dependency relationship between two higher priority tasks. Specifically, the higher priority task in a task pair is not allowed to execute if the lower priority task in the pair is released even if there are idle processors. We propose a method to properly select task pairs until all tasks are schedulable. To test the system schedulability, we provide an analysis method to bound the response time of each task under the constraint of task pairs. In the simulation study, we generate random task sets and compare the acceptance ratio of our method and GSYY. The results shows that our method outperforms GSYY method as expected. As future work, we will study how to efficiently select the optimal combinations of task pairs to further improve the performance.

References

1. M. Kamruzzaman, S. Swanson and D. M. Tullsen, Inter-core prefetching for multicore processors using migrating helper threads, *ACM SIGPLAN Not.* **46** (2011) 393–404.
2. A. Sarkar, F. Mueller and H. Ramaprasad, Predictable task migration for locked caches in multi-core systems, *ACM SIGPLAN Not.* **46** (2011) 131–140.
3. J. Zhou and T. Wei, Stochastic thermal-aware real-time task scheduling with considerations of soft errors, *J. Syst. Softw.* **102** (2015) 123–133.
4. J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu and Y. Ma, Thermal-aware task scheduling for energy minimization in heterogeneous real-time mpsoc systems, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **35** (2016) 1269–1282.
5. J. Zhou, K. Cao, P. Cong, T. Wei, M. Chen, G. Zhang, J. Yan and Y. Ma, Reliability and temperature constrained task scheduling for makespan minimization on heterogeneous multi-core platforms, *J. Syst. Softw.* **133** (2017) 1–16.
6. T. P. Baker and M. Cirinei, Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks, *OPODIS*, 2007, pp. 62–75.
7. N. Guan, Q. Deng, Z. Gu, W. Xu and G. Yu, Schedulability analysis of preemptive and nonpreemptive edf on partial runtime-reconfigurable fpgas, *ACM Trans. Des. Autom. Electron. Syst.* **13** (2008) 56:1–56:43.
8. G. Geeraerts, J. Goossens and M. Lindstrm, Multiprocessor schedulability of arbitrary-deadline sporadic tasks: Complexity and antichain algorithm, *Real-Time Syst.* **49** (2013) 171–218.
9. Y. Sun, G. Lipari, N. Guan and W. Yi, Improving the response time analysis of global fixed-priority multiprocessor scheduling, *Proc. IEEE 20th Int. Conf. Embedded and Real-Time Computing Systems and Applications*, August 2014, pp. 1–9.
10. S. Baruah and N. Fisher, Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems, *Princ. Distrib. Syst.* (2007) 204–216.

11. R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters and M. Bertogna, Global and partitioned multiprocessor fixed priority scheduling with deferred preemption, *ACM Trans. Embed. Comput. Syst.* **14** (2015) 47:1–47:28.
12. T. Zhang, N. Guan, Q. Deng and W. Yi, Start time configuration for strictly periodic real-time task systems, *J. Syst. Arch.* **66** (2016) 61–68.
13. T. P. Baker, Multiprocessor edf and deadline monotonic schedulability analysis, *RTSS*, 2003, pp. 120–129.
14. M. Bertogna, M. Cirinei and G. Lipari, Improved schedulability analysis of edf on multiprocessor platforms, *ECRTS*, 2005, pp. 209–218.
15. M. Bertogna and M. Cirinei, Response-time analysis for globally scheduled symmetric multiprocessor platforms, *RTSS*, 2007.
16. S. K. Baruah, Techniques for multiprocessor global schedulability analysis, *RTSS*, 2007, pp. 119–128.
17. J. Lee, Time-reversibility of schedulability tests, *Proc. IEEE Real-Time Systems Symp.* December 2014, pp. 294–303.
18. N. Guan, M. Han, C. Gu, Q. Deng and W. Yi, Bounding carry-in interference to improve fixed-priority global multiprocessor scheduling analysis, *Proc. IEEE 21st Int. Conf. Embedded and Real-Time Computing Systems and Applications*, August 2015, pp. 11–20.
19. N. Guan, M. Stigge, W. Yi and G. Yu, New response time bounds for fixed priority multiprocessor scheduling, *RTSS*, 2009, pp. 387–397.
20. R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters and M. Bertogna, Global fixed priority scheduling with deferred pre-emption, *Proc. IEEE 19th Int. Conf. Embedded and Real-Time Computing Systems and Applications*, August 2013, pp. 1–11.
21. E. Bini and G. C. Buttazzo, Biasing effects in schedulability measures, in *Proc. 16th Euromicro Conf. Real-Time Systems ECRTS 2004*, June 2004, pp. 196–203.