Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms

Risat Pathan, Petros Voudouris, and Per Stenström

Abstract—We consider the scheduling of a real-time application that is modeled as a collection of parallel and recurrent tasks on a multicore platform. Each task is a directed-acyclic graph (DAG) having a set of subtasks (i.e., nodes) with precedence constraints (i.e., directed edges) and must complete the execution of all its subtasks by some specified deadline. Each task generates potentially infinite number of instances where the releases of consecutive instances are separated by some minimum inter-arrival time. Each DAG task and each subtask of that DAG task is assigned a fixed priority. A two-level preemptive global fixed-priority scheduling (`GFP`) policy is proposed: a *task-level scheduler* first determines the highest-priority ready task and a *subtask-level scheduler* then selects its highest-priority subtask for execution. To our knowledge, no earlier work considers a two-level `GFP` scheduler to schedule recurrent DAG tasks on a multicore platform. We derive a *schedulability test* for our proposed two-level `GFP` scheduler. If this test is satisfied, then it is guaranteed that all the tasks will meet their deadlines under `GFP`. We show that our proposed test is not only theoretically better but also empirically performs much better than the state-of-the-art test in scheduling randomly generated parallel DAG task sets.

Index Terms—Real-Time Systems, Parallel DAG Tasks, Global Fixed-priority Scheduling, Schedulability Analysis, Multicore Processors

1 Introduction

The increasing demand for more advanced functions in todays prevailing time-critical systems is, and will be, met using computation power of multicore processors. One of the main challenges for such systems is to maximize the utilization of the parallel multicore architecture while meeting the real-time deadlines of the application tasks.

Sequential programming has been the primary paradigm to implement real-time tasks on uni- and multicore platforms [1]. However, the sequential paradigm does not allow *intra-task parallelism*: each task must execute sequentially, which limits the extent to which processing capacity of a parallel architecture can be exploited (according to Amdahl's law [2]). On the other hand, the high-performance computing community has developed several parallel programming models – task parallelism (e.g., Cilk [3]), data parallelism (e.g., OpenMp loops [4]) — to better exploit the parallel multicore architecture. Parallel programming paradigms allow both inter- and intra-task parallelism: each classical sequential task is implemented as a collection of parallel subtasks that can execute in parallel on multiple cores.

Many of the the parallel applications like Sort, Strassen, and FFT in the Barcelona OpenMP Task Suites are widely used in many real-time applications [5]. The work in [6] considers a 3D multigrid solver which is implemented as a parallel task for executing on MERASA multicore platform. During the last couple of years, researchers have proposed real-time scheduling algorithms and schedulability analysis of several parallel task models for multicore architectures [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. The directed acyclic graph (DAG) based task model is the most general parallel task model proposed in the literature. A DAG task consists of a set of nodes and directed edges where each node is a sequential subtask and each directed edge is a precedence constraint between two subtasks.

Tasks on a multicore platform can be scheduled statically or dynamically. In static scheduling (also known as partitioned scheduling) of a parallel task, the subtasks are statically pre-assigned to fixed cores [17] which may severely underutilizes hardware resources due to load imbalance or communication overheads. On the other hand, dynamic scheduling (also known as global scheduling) in which the subtasks are allowed to execute on any core can significantly improve resource utilization [18]. However, the schedulability analysis to guarantee that all the parallel tasks meet their deadlines under dynamic scheduling paradigm is more complex due to many possible interleavings of the threads across the cores. Therefore, overly pessimistic assumptions are made to analyze parallel real-time DAG tasks which does not allow to enjoy the full speedup on a multicore architecture.

This paper considers dynamic (i.e., global) scheduling of a collection of recurrent DAG tasks. Each recurrent DAG task generates potentially infinite DAG instances where two consecutive instances are separated by some minimum inter-arrival time (also called the period). Each DAG task and each subtask of that DAG task are assigned fixed priorities. The tasks are scheduled using a *two-level preemptive global fixed-priority* (`GFP`) scheduler. A task-level scheduler first determines the highest-priority ready task G_{hp} and a subtask-level scheduler then selects the highest-priority subtask of G_{hp} for execution. If a lower-priority subtask is in execution while all the cores are busy, a newly released higher-priority subtask preempts the execution of the lower-priority subtask. The `GFP` scheduling allows a subtask to execute on *any* core of a multicore processor even when it resumes execution after being preempted by a higher-priority subtask.

To apply a particular real-time scheduling algorithm in the

• *R. Pathan, P. Voudouris, P. Stenström are with the Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96, Sweden. E-mail: {risat, petrosv, pers}@chalmers.se*

Manuscript received –, –; revised – .

domain of real-time safety-critical systems like automotive or aerospace, the system designer needs to verify offline (i.e., before the system is in mission) that all the timing constraints are met. The major tool that is used to do such verification is offline analysis of the scheduling algorithm to derive a schedulability test for the two-level GFP scheduler. If this test is satisfied for a given task set and a processing platform, it is guaranteed that all the timing constraints will also be met during actual runtime. The main endeavor of this paper is to present the schedulability analysis of our proposed two-level GFP scheduling algorithm to derive a schedulability test.

One of the major sources of pessimism for the analysis of global GFP scheduling is the computation of intra-task and inter-task interference that a subtask under analysis suffers from its higher-priority subtasks due to the complex internal structures of the DAG tasks (e.g., number of parallel subtasks, precedence constraints, etc). No work on GFP scheduling of DAG tasks exploits the internal structures of individual DAG task to determine both intra- and inter-task interference. Melani et al. [16] recently proposed a response-time[1] based schedulability analysis for GFP scheduling of recurrent DAG tasks. However, the work in [16] does not consider any particular subtask-level scheduler and hides the internal structures of the DAG tasks by embracing pessimism in their analysis, which in turn requires a relatively larger number of cores to meet the deadlines of all the tasks (it will be demonstrated in our experimental Section 6).

It is worth mentioning at this point that the work in [16] considers a relative more general DAG task model (known as conditional DAG tasks) in which some subtasks of a DAG task may or may not be generated depending on some conditional (e.g., if-then-else) statement. In other words, the DAG structures of different instances of a conditional DAG task may be different while the same DAG is generated for all the instances of a non-conditional DAG task. We learned that the degree of pessimism in the schedulability analysis in [16] is not due to the consideration of a more general DAG task model rather such pessimism equally applies for the analysis of non-conditional DAG tasks. To that end we consider non-conditional DAG tasks in this paper and show the effectiveness of our analysis in significantly reducing the pessimism of the analysis in [16] even for such non-conditional DAG tasks. Extending our analysis for conditional DAG tasks is an interesting future work.

There are many works in non-real-time systems domains that consider assigning priorities to the subtasks with the main aim of reducing the average completion time [3], [19], [20], [21]. For example, the HEFT algorithm [21] assigns higher priority to a subtask with a relatively higher upward rank. However, the HEFT algorithm is designed for improving the average-case performance and the completion time of DAG may be very large for some not-so-frequent (i.e., worst) case and could miss the deadline of the task. This paper proposes scheduling algorithm and its analysis considering the worst-case, i.e., a safe upper bound on the completion time of the DAG can be computed where subtasks are assigned priorities, which is a major contribution of this paper.

This paper has the following contributions. *First*, we consider a specific subtask-level GFP scheduler to schedule the subtasks to efficiently utilize the processing cores and to reduce the pessimism in schedulability analysis in comparison to earlier approaches. A simple but very effective heuristic to assign the fixed priorities to the subtasks of each DAG task is proposed by unfolding the internal structures (i.e., topology) of the DAG tasks. *Second*, we propose new techniques to compute both intra- and inter-task interference. To this end, we propose a new response-time-based schedulability test for GFP scheduling of recurrent DAG tasks. *Third*, we demonstrate the effectiveness of our proposed test based on empirical study. Experimental results using randomly generated DAG task sets show that our proposed test outperforms the state-of-the-art test for various (i) loads/utilizations of the application, (ii) number of cores, and (iii) number of tasks of the application.

The rest of the paper is organized as follows. The system model and important definitions are presented in Section 2. The overview of our schedulability analysis framework is presented in Section 3. The details of the schedulability analysis of the two-level GFP scheduler is presented in Section 4. The pessimism of the state-of-the-art test is presented in Section 5 using an example. We present our results in Section 6. Related works are presented in Section 7 before we conclude in Section 8.

2 System Model and Useful Definitions

We consider the preemptive GFP scheduling of n recurrent DAG tasks in set $\Gamma = \{G_1, \ldots G_n\}$ on a multicore processor having m cores. The (normalized) speed of each core is 1. Each DAG task $G_k \in \Gamma$ generates potentially infinite number of DAG instances, called the *jobs* of the task, such that the releases of two consecutive jobs of G_k is separated by a minimum distance. Each DAG task G_k is characterized using four parameters $G_k = (T_k, D_k, V_k, E_k)$ where

- T_k is the minimum inter-arrival time of consecutive jobs (also, called the period) of task G_k;
- D_k is the relative deadline such that $D_k \leq T_k$;
- $V_k = \{v_{k,1}, \ldots v_{k,n_k}\}$ is a set of n_k subtasks[2]; and
- $E_k \subseteq (V_k \times V_k)$ is a set of directed arcs (or edges).

The parameter D_k specifies the *real-time constraint*: if the source (i.e., the first) subtask of some job of task G_k becomes ready for execution at time t, then the execution of all the subtasks of that job must complete by time $(t + D_k)$. Each subtask $v_{k,j} \in V_k$ represents a sequential chunk of execution having worst-case execution time (WCET) equal to $C_{k,j}$. If $(v_{k,p}, v_{k,q}) \in E_k$, then subtask $v_{k,q}$ can start execution after subtask $v_{k,p}$ completes its execution. An example of a DAG task G_k is shown in Figure 1 where $T_k = 100$ and $D_k = 52$.

Let the task G_k in Figure 1 first arrives at time x. In other words, the first job or instance of this DAG task G_k becomes ready for execution at time x. This job has a deadline $(x + D_k) = (x + 52)$ by which all its subtasks must finish their execution. The second job of task G_k is released no earlier than time $(x + T_k) = (x + 100)$ since the minimum inter-arrival time is $T_k = 100$. Although the second job of the task G_k is released no earlier than time $(x + 100)$, its actual release time may be later than $(x + 100)$ because we are considering sporadic tasks. Let the second job is released at time $(x + 120)$. Then the deadline of the second job is $D_k = 52$ time units after its release time, i.e., at time $(x + 120) + 52 = (x + 172)$. The third job of this task is released no earlier than time $(x + 220)$, and so on. Note that the deadline is specified for the entire job of the task, i.e., the execution of all the subtasks must be completed before that job's deadline.

1. The response time of a DAG task is also known as the makespan, which is the longest possible time any instance of the task requires to complete its execution.

2. There are works where the terms "workflow" and "task" are respectively used to mean "task" and "subtask" as are used in this paper.

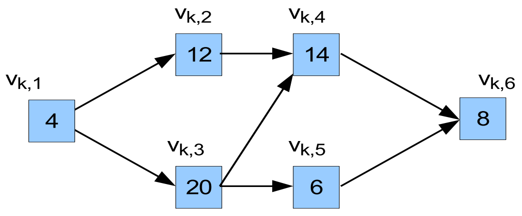


Figure 1: A DAG task G_k with six subtasks. The WCET of each subtask is shown inside each shape. The priority ordering is $v_{k,1}$(highest) $\succ v_{k,3} \succ v_{k,2} \succ v_{k,5} \succ v_{k,4} \succ v_{k,6}$(lowest).

A subtask with no incoming (res. outgoing) edge is called a *source* (resp. *sink*) subtask. Without loss of generality we assume that there is exactly one source (denoted as v_k^{src}) and one sink (denoted as v_k^{sink}) of G_k. If there is more than one source (sink) subtask, a dummy subtask with WCET equal to zero as a new source (sink) subtask is added with arcs to (from) all actual source (sink) subtasks.

We define a *path* in the DAG of task G_k originating at subtask $v_{k,a}$ as a sequence of subtasks $(v_{k,a}, \ldots v_{k,b})$ such that $(v_{k,j}, v_{k,j+1}) \in E_k$, $a \le j < b$. The length of a path is the sum of the WCETs of the subtasks in the path. The *longest path* of G_k is denoted by L_k which is the length of the longest path in G_k. For example, $L_k = 46$, which corresponds to the path $(v_{k,1}, v_{k,3}, v_{k,4}, v_{k,6})$ in Figure 1.

We denote by W_k the *total work* of task G_k which is equal to the sum of WCETs of all the subtasks of V_k, i.e., $W_k = \sum_{j=1}^{n_k} C_{k,j}$. For example, $W_k = 64$ in Figure 1.

For each subtask $v_{k,j} \in V_k$, the *ancestors* of $v_{k,j}$, denoted by $Ancst_k^j$, is the set of subtasks $v_{k,p} \in V_k$ such that there exists a path from $v_{k,p}$ to $v_{k,j}$. Subtask $v_{k,j} \in V_k$ becomes ready for execution when each of the subtasks in $Ancst_k^j$ completes execution (i.e., all dependencies are released). The values of L_k, W_k, and the set $Ancst_k^j$ can be computed in polynomial time in the representation of G_k [22].

Task-level Priority Assignment. The fixed priorities of the n tasks $\{G_1, G_2, \ldots G_n\}$ are assigned based on Deadline-Monotonic (DM) priority ordering (i.e., a task having shorter relative deadline has higher fixed priority). The set of tasks having higher fixed priorities than that of task G_k is denoted by $\mathtt{hpt}_k$.

Subtask-level Priority Assignment. The fixed priorities to the subtasks of G_k are assigned based on a topological order of the subtasks of G_k. A topological order is such that if there is an edge from subtask $v_{k,j}$ to subtask $v_{k,g}$, then subtask $v_{k,j}$ appears before subtask $v_{k,g}$ in the topological order. A topological order can be computed in linear time in the size of the DAG of G_k [22]. A subtask $v_{k,j} \in V_k$ is assigned a higher fixed priority than subtask $v_{k,g} \in V_k$ if and only if $v_{k,j}$ appears before subtask $v_{k,g}$ in the topological order of G_k. This subtask-level priority assignment policy exploits the internal structure of the DAG task G_k based on its topology.

The topological sort of DAG G_k in Figure 1 is given as follows ($u \succ w$ implies u has higher priority than w):

$$v_{k,1} \succ v_{k,3} \succ v_{k,2} \succ v_{k,5} \succ v_{k,4} \succ v_{k,6}$$

During the subtask-level priority assignment, a subtask at a lower level of the DAG is given higher priority in comparison to the one at a higher level. If two subtasks have the same level, then the subtask having relatively higher (subtask) index is given higher priority.

Given a subtask $v_{k,j} \in V_k$ of particular task G_k, the set of subtasks in V_k having higher fixed priorities than that of $v_{k,j}$ is denoted by $\mathtt{hpst}_k^j$.

The two-level GFP Scheduler. The subtasks of the DAGs are executed based on a two-level scheduler. The task-level scheduler determines the highest-priority ready task G_{hp} and the subtask-level scheduler selects the highest-priority subtask of G_{hp} for execution. A newly released relatively higher-priority subtask is allowed to preempt the execution of a lower-priority subtask when all the cores are busy.

Under the subtask-level fixed-priority assignment policy, subtask $v_{k,j}$ has lower fixed priority than all the subtasks in set $Ancst_k^j$. All the ancestors of $v_{k,j}$ are also in the set of higher priority subtasks of $v_{k,j}$, i.e., $Ancst_k^j \subseteq \mathtt{hpst}_k^j$ since subtasks are assigned priorities based on topological sort. For example, for the subtask $v_{k,4}$ in Figure 1 we have $Ancst_k^4 = \{v_{k,1}, v_{k,2}, v_{k,3}\}$ and $\mathtt{hpst}_k^4 = \{v_{k,1}, v_{k,2}, v_{k,3}, v_{k,5}\}$. Note that $Ancst_k^4 \subseteq \mathtt{hpst}_k^4$.

In this paper, we determine the response time (also known as makespan) of each task $G_k \in \Gamma$. The response time of a task G_k is the largest possible time that all the subtasks of *any* job of task G_k requires to finish its execution relative to the time when the source subtask of the job becomes ready for execution. The response time of task G_k is denoted as R_k. We compute R_k based on computing the response time of each subtask $v_{k,j} \in V_k$ of task G_k. The response time of subtask $v_{k,j}$ of task G_k is denoted as R_k^j.

3 Overview of Our Analysis Framework

This section presents an overview of our schedulability analysis framework that is used to derive a *schedulability test*. If this test is when satisfied, then it is guaranteed that all the tasks meet their deadlines (i.e., all the real-time constraints are met). The framework is similar to that of in [16]. To determine whether each task $G_k \in \Gamma$ meets its deadlines or not, the response time R_k^j of each subtask $v_{k,j} \in V_k$ of an *arbitrary job* of task G_k is computed based on worst-case schedulability analysis.

The schedulability analysis of each subtask $v_{k,j}$ is performed in an interval $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$, called the *problem window* of length t, such that the subtask $v_{k,j}$ is released (i.e., becomes ready for execution) at time $\mathtt{rdy}_k^j$ relative to the release time of the source subtask of the arbitrary job. A subtask is said to be released when all its ancestors have completed their execution (i.e., there is no dependency). The problem window of $v_{k,j}$ is depicted in Figure 2.

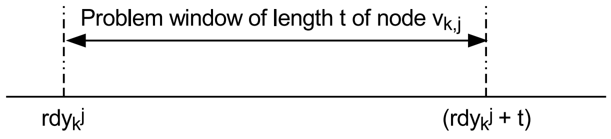


Figure 2: Problem window $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$ of subtask $v_{k,j}$ that becomes ready for execution at time $\mathtt{rdy}_k^j$ relative to the release time of the source subtask.

Within the problem window $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$, the execution of subtask $v_{i,j}$ may be interfered by the higher priority subtasks in set $\mathtt{hpst}_k^j$, and also by the subtasks of the higher-priority tasks in set $\mathtt{hpt}_k$. The response time R_k^j of $v_{k,j}$ is derived by computing the *total interfering workload* and *interference*. Before computing these terms, their definitions are formally presented.

Total Interfering Workload. The *intra-task interfering workload* of a higher priority subtask $v_{k,h} \in \mathtt{hpst}_k^j$ in the problem window of $v_{k,j}$ is the cumulative length of intervals during which $v_{k,h}$ executes in that window. Similarly, the *inter-task interfering workload* of a higher-priority task $G_i \in \mathtt{hpt}_k$ in the problem window of $v_{k,j}$ is the cumulative length of intervals during which the subtasks of the jobs of task G_i execute in that window. Since computing the exact interfering workload considering all possible release times of the tasks is computationally infeasible, we will determine an upper bound on the intra- and inter-task interfering workloads. The *total interfering workload* is the sum of intra- and inter-task interfering workloads.

Interference. The *interference* on subtask $v_{k,j}$ within its problem window is the cumulative length of the intervals during which subtask $v_{k,j}$ is ready but *not* executing. Under GFP scheduling, the execution of subtask $v_{k,j}$ in an interval $[a, b]$ is interfered only if all the m processing cores are *simultaneously busy* executing the higher-priority subtasks (i.e., subtasks in $\mathtt{hpst}_k^j$ and/or subtasks of the tasks in $\mathtt{hpt}_k$) in $[a, b]$. The interference is calculated based on total interfering workload as follows.

A note on the relationship between total interfering workload and interference. Let $\mathcal{I}_{k,j}(t)$ is the total interfering workload due to the execution of the higher-priority subtasks in $\mathtt{hpst}_k^j$ and the subtasks of the higher priority tasks in $\mathtt{hpt}_k$ within the problem window of length t of subtask $v_{k,j}$. By considering the worst-case in which the total interfering workload $\mathcal{I}_{k,j}(t)$ in the problem window keeps all the m cores simultaneously busy (i.e., no core is idle), the maximum interference that the subtask $v_{k,j}$ suffers in its problem window of length t is $\mathcal{I}_{k,j}(t)/m$.

The difference between the length of the problem window and interference in the problem window (i.e., $t - \mathcal{I}_{k,j}(t)/m$) is the cumulative length of intervals in the problem window during which there is *at least* one core on which subtask $v_{k,j}$ can execute. If the sum of interference and the WCET of subtask $v_{k,j}$ is not larger than the length of the problem window, then its is guaranteed that the subtask $v_{k,j}$ can complete its execution in the problem window.

4 Response Time Computation

This section presents the schedulability analysis of a DAG task $G_k \in \Gamma$ to determine its response time R_k. If $R_k \leq D_k$, then it is guaranteed that all the jobs of G_k meet their deadlines. To compute R_k, the response time R_k^j of each subtask $v_{k,j} \in V_k$ of task G_k is computed. The response time R_k is equal to the response time of the sink subtask v_k^{sink} and is given as follows:

$$R_k = \max_{v_{k,j} \in V_k} \{R_k^j\} = R_k^{sink} \tag{1}$$

4.1 Computing R_k^j for subtask $v_{k,j}$

The response time of the higher-priority tasks in $\mathtt{hpt}_k$ is computed before computing the response time of (the lower-priority) task G_k. We also compute the response times of the subtasks of G_k in their *decreasing* priority order, i.e., the response time of the source subtask (i.e., the highest priority subtask) is computed first and the response time of the sink subtask (i.e., the lowest priority subtask) of G_k is computed the last. Therefore, the response time of $v_{k,j}$ is computed only after the response times of all its ancestor[3] subtasks in $Ancst_k^j$ are computed. To determine the response time R_k^j of subtask $v_{k,j}$, we follow the following four steps:

- Step 1 (Find the Ready Time): We first determine the *latest time* when subtask $v_{k,j}$ becomes ready (i.e, value of $\mathtt{rdy}_k^j$) for execution relative to the release time of the source subtask $v_{k,src}$ of any (arbitrary) job of G_k. By considering that the subtask $v_{k,j}$ must have a response time not smaller than t, the schedulability analysis of $v_{k,j}$ to compute R_k^j is performed within its problem window $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$ of length t. If subtask $v_{k,j}$ does not complete within $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j+t)$, then the length t of the problem window is increased until subtask $v_{k,j}$ is guaranteed to complete within $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$ or the deadline is missed (this is a well-known approach to compute response time, for example, in [16]).
- Step 2 (Find the Intra-Task Interfering Workload). We determine the maximum interfering workload of the *higher-priority subtasks* in set $\mathtt{hpst}_k^j$ on subtask $v_{k,j}$ in the problem window $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$.
- Step 3 (Find the Inter-Task Interfering Workload). We determine the maximum interfering workload of the subtasks of the jobs of the *higher-priority tasks* in set $\mathtt{hpt}_k$ on subtask $v_{k,j}$ in the problem window $[\mathtt{rdy}_k^j, \mathtt{rdy}_k^j + t)$.
- Step 4 (Find Interference and the Response Time). Based on the intra- and inter-task interfering workloads, we determine the maximum interference that subtask $v_{i,j}$ suffers in its problem window. Finally, based on the interference we determine the response time R_k^j.

Step 1 (Find the Ready time): Consider that an arbitrary job of task G_k is released at time r_k, i.e., the source subtask $v_{k,src}$ of the job becomes ready at time r_k. In other words, the latest time by which the source subtask $v_{k,src}$ of the job of G_k becomes ready for execution relative to time r_k is 0. Therefore, we have $\mathtt{rdy}_k^{src} = 0$.

A (non-source) subtask $v_{k,j}$ becomes ready for execution after all the ancestor subtasks in $Ancst_k^j$ complete their execution (i.e., when all of its dependencies are released). Therefore, the *latest time* when $v_{k,j}$ becomes ready for execution is the maximum

3. Recall from the subtask-level priority assignment policy in Section 2 that the ancestors of $v_{k,j}$ have higher priority than the priority of $v_{k,j}$.

response times of all its ancestors. The latest ready time $\texttt{rdy}_k^j$ of subtask $v_{k,j}$ is:

$$\texttt{rdy}_k^j = \max_{v_{k,a} \in Ancst_k^j} R_k^a \tag{2}$$

Since the response times of all the ancestor subtasks of $v_{k,j}$ are computed before computing the response time of $v_{k,j}$, the value of $\texttt{rdy}_k^j$ in Eq. (2) can be computed based on the (already available) response times of all its ancestors.

Step 2 (Find the Intra-Task Interfering Workload). In this step, we determine the maximum interfering workload of the higher-priority subtasks in set $\texttt{hpst}_k^j$. None of the ancestors of $v_{k,j}$ can interfere the execution of $v_{k,j}$ within the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ since all the ancestors of $v_{k,j}$ must have completed by time $\texttt{rdy}_k^j$ (according to Eq. (2)). In other words, the higher-priority subtasks from set $\texttt{hpst}_k^j$ that may interfere the execution of subtask $v_{k,j}$ are in set $\mathcal{S}_k^j$ which is given as follows:

$$\mathcal{S}_k^j = \texttt{hpst}_k^j - Ancst_k^j \tag{3}$$

The intra-task interfering workload in the problem window is now computed based on the interfering workload of each of the subtasks in $\mathcal{S}_k^j$. Since $\texttt{rdy}_k^j$ is the time when subtask $v_{k,j}$ becomes ready for execution, a higher-priority subtask $v_{k,h} \in \mathcal{S}_k^j$ executes within the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ $\underline{\text{only if}}$ $v_{k,h}$ has not completed its execution by time $\texttt{rdy}_k^j$. Recall that for each higher-priority subtask $v_{k,h} \in \mathcal{S}_k^j$, the response time R_k^h has already been computed before computing $\texttt{rdy}_k^j$.

If $R_k^h > \texttt{rdy}_k^j$, then the higher-priority subtask $v_{k,h}$ completes its execution after time $\texttt{rdy}_k^j$ and contributes at most $(R_k^h - \texttt{rdy}_k^j)$ as the intra-task interfering workload within the problem window. Otherwise, we have $R_k^h \leq \texttt{rdy}_k^j$ which implies that subtask $v_{k,h}$ has already completed its execution by time $\texttt{rdy}_k^j$ and cannot contribute to the intra-task interfering workload. Therefore, subtask $v_{k,h} \in \mathcal{S}_k^j$ can execute at most $max\{0, R_k^h - \texttt{rdy}_k^j\}$ time units inside the problem window of $v_{k,j}$. However, since the WCET of subtask $v_{k,h}$ is $C_{k,h}$, the quantity $max\{0, R_k^h - \texttt{rdy}_k^j\}$ can be upper bounded by $C_{k,h}$. Consequently, the higher-priority subtask $v_{k,h} \in \mathcal{S}_k^j$ can execute for at most $min\{C_{k,h}, max\{0, R_k^h - \texttt{rdy}_k^j\}\}$ time units inside the problem window of $v_{k,j}$. The total intra-task interfering workload, denoted by $\mathcal{W}_k^{intra}$, due to all the higher-priority subtasks in set $\mathcal{S}_k^j = (\texttt{hpst}_k^j - Ancst_k^j)$ is

$$\mathcal{W}_k^{intra} = \sum_{v_{k,h} \in \mathcal{S}_k^j} min\{C_{k,h}, max\{0, R_k^h - \texttt{rdy}_k^j\}\} \tag{4}$$

Step 3 (Find the Inter-Task Interfering Workload). We now determine the maximum interfering workload of the higher-priority tasks in set $\texttt{hpt}_k$ within the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ of length t. Given a problem window of length t, we denote[4] by $\mathcal{W}_{k,t}^{inter}$ the maximum inter-task interfering workload of the jobs of the higher priority tasks in $\texttt{hpt}_k$. To determine $\mathcal{W}_{k,t}^{inter}$, we compute the inter-task interfering workload of each individual higher-priority task $G_i \in \texttt{hpt}_k$ within the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ of subtask $v_{k,j}$.

The execution of different jobs of task $G_i \in \texttt{hpt}_k$ in the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ of subtask $v_{k,j}$ is divided in to three categories: carry-in job, body jobs, and carry-out job. The *carry-in job* is the first job of task G_i that executes in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ such that its release time is before $\texttt{rdy}_k^j$ and its deadline is in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$. The *carry-out job* is the last job of task G_i that executes in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ such that its release time is before $(\texttt{rdy}_k^j + t)$ and deadline is after $(\texttt{rdy}_k^j + t)$. All other jobs of task G_i executing in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$ are *body jobs*.

To determine the maximum inter-task interfering workload of the jobs of the higher priority task $G_i \in \texttt{hpt}_k$ inside the problem window, we need to consider the *worst-case release pattern* of the carry-in, body, and carry-out jobs of task G_i such that their contribution to the inter-task interfering workload is maximized. The worst-case release pattern refers to the scenario in which the releases and execution of the jobs of the higher priority task $G_i \in \texttt{hpt}_k$ inflict maximum interference on a lower priority task inside its problem window. We consider the following worst-case release pattern that is depicted in Figure 3 where

- the carry-out job[5] starts its execution at time $(\texttt{rdy}_k^j + t - W_i/m)$ and completes W_i amount of work by time $(\texttt{rdy}_k^j + t)$;
- all the earlier jobs *arrive* as late as possible (i.e., strictly periodically); and
- the carry-in job of G_i finishes its execution at time $(\texttt{rdy}_k^j + t_{cin})$ such that $(\texttt{rdy}_k^j + t_{cin} - R_i)$ is the release time of the carry-in job of G_i for some $t_{cin} > 0$ where R_i is the (already computed) response time of task G_i.

Similar to the body and carry-out jobs, we may also consider that the carry-in job executes W_i amount of work in W_i/m time units and completes at time $(\texttt{rdy}_k^j + t_{cin})$, which is an upper bound on the work done by the carry-in job in the problem window. However, we will compute the workload of the carry-in job more precisely by exploring the internal structure of G_i. Therefore, the work done by the carry-in job, unlike the body and carry-out job, is not depicted in Figure 3 to complete W_i amount of work in a duration equal to W_i/m.

Before we present the mathematical expressions to compute the inter-task interfering workload of task $G_i \in \texttt{hpt}_k$ in the problem window based on Figure 3, we first prove in Lemma 1 that the release pattern in Figure 3 is in fact the worst-case, i.e., it maximizes the inter-task interfering workload of task G_i in the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$.

Lemma 1. *An upper bound on the interfering workload of task G_i in a problem window of length t can be computed based on the worst-case release pattern of the carry-in, body and carry-out jobs of G_i that is given in Figure 3.*

Proof. We prove this lemma by showing that shifting the problem window in Figure 3 either to the right or to the left cannot increase workload in the problem window.

Consider shifting the window to the right where the arrival patter remains the same and we move $[\texttt{rdy}_k^j$ and $\texttt{rdy}_k^j + t]$ to the

4. Note that the right-hand side of Eq. (4) is independent of the length of the problem window, i.e., value of t. Therefore, the notation $\mathcal{W}_k^{intra}$ does not include the symbol "t". On the other hand (as it will be evident shortly) the total inter-task interfering workload depends on the length of the problem window t and we include the symbol "t" in $\mathcal{W}_{k,t}^{inter}$.

5. Completing W_i amount of work in W_i/m time units contributes to maximum work of the carry-out job in the least possible length of the problem window, which leaves maximum length for the execution of the carry-in and carry-out job. This leads us to compute the worst-case workload of G_i.

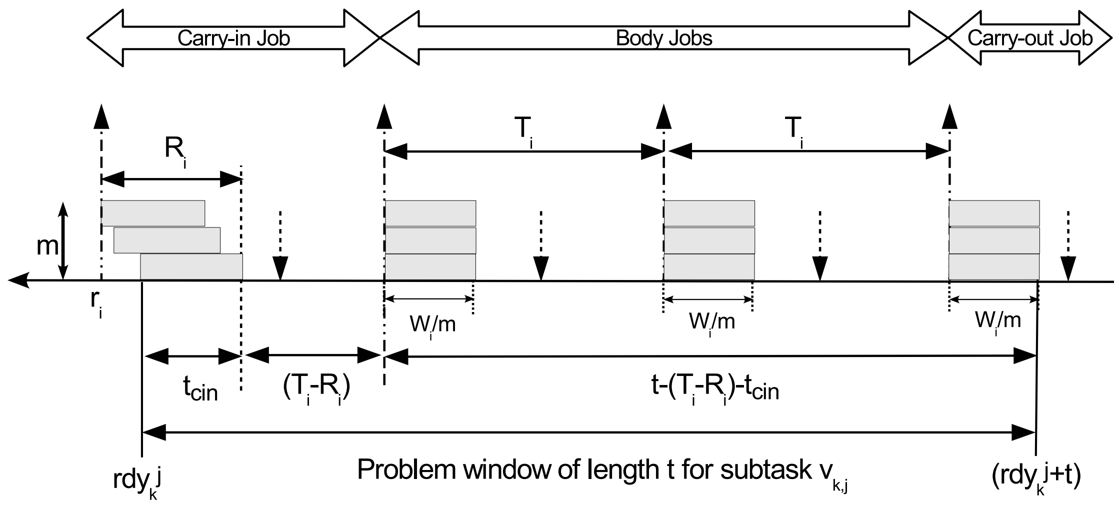


Figure 3: Worst-case release pattern. The (larger) upward dotted arrow and (shorter) downward dotted arrow respectively represents the release time and the deadline of a job.

right. Note that when a body job becomes a carry-in job during such right shift, we consider its execution similar to the carry-in job shown in the non-shifted window in Figure 3. Shifting the window to the right by δ time units is same as shifting the window to the left by $(T_i - (\delta \ mod \ T_i))$ time units. Therefore, we only consider left shifting the problem window.

Consider shifting the window in Figure 3 to left by δ time units where $\delta \leq W_i/m$. By shifting we mean that we keep the arrival pattern of the jobs unchanged but the problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t]$ is moved leftward. In such case, the workload is reduced by $(\delta \cdot m)$ from right side of the window and at most $(\delta \cdot m)$ amount of work may be increased from the left. Shifting the window to the left by δ time units, where $W_i/m < \delta < T_i$, would reduce at least W_i amount of work from the right and the amount of work that can be increased from the left is at most W_i. This is because at most W_i amount of additional work can be generated from the left in an interval not larger than T_i in Figure 3. Shifting the window to the left by δ time units, where $\delta \geq T_i$, produces scenario equivalent to shifting the window left by $(\delta \ mod \ T_i)$ time units, which has already been considered. Therefore, the workload cannot increase for any possible shift of the problem window. □

According to Figure 3, the carry-in and the body jobs execute in the interval $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t - W_i/m]$. The length of the interval $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t - W_i/m]$ is given (denoted by $\mathcal{X}_i(t)$) as follows:

$$\mathcal{X}_i(t) = max\{0, \texttt{rdy}_k^j + t - \frac{W_i}{m}\} \tag{5}$$

where $\texttt{rdy}_k^j$ is already computed using Eq. (2).

Next we will determine the value of t_{cin} which is the length of the interval where the carry-in job of τ_i executes in the problem window. The maximum number of body jobs that execute in the problem window is $\lfloor \frac{\mathcal{X}_i(t)}{T_i} \rfloor$. Therefore, the length of the interval inside the problem window where all the body jobs execute is $(\lfloor \frac{\mathcal{X}_i(t)}{T_i} \rfloor \cdot T_i)$. The length of the interval inside the problem window where the carry-out job executes is W_i/m.

According to Figure 3, the carry-in job completes its execution at time $(\texttt{rdy}_k^j + t_{cin})$. Note that there is an interval of length $(T_i - R_i)$, just after the time instant $(\texttt{rdy}_k^j + t_{cin})$, during which no job of G_i can be released since the releases of two consecutive jobs of G_i is separated by at least the period T_i of task G_i. Therefore, the length of the interval inside the problem window where the carry-in job execute, (i.e., value of t_{cin}) is computed as follows:

$$t_{cin} = t - (T_i - R_i) - \left\lfloor \frac{\mathcal{X}_i(t)}{T_i} \right\rfloor \cdot T_i - \frac{W_i}{m} \tag{6}$$

To avoid negative length of t_{cin} in Eq. (6), we lower bound the value of t_{cin} by zero and t_{cin} is given as follows:

$$t_{cin} = max\left\{0, t - (T_i - R_i) - \left\lfloor \frac{\mathcal{X}_i(t)}{T_i} \right\rfloor \cdot T_i - \frac{W_i}{m}\right\} \tag{7}$$

Note that if $t_{cin} = 0$, then there is no carry-in job. The only jobs of G_i that execute inside the problem window are the body jobs and the carry-out job.

Now we compute the inter-task interfering workload of the carry-in, body and carry-out jobs. We denote by $\texttt{CR}_i(t_{cin})$ the maximum inter-task interfering workload of the *carry-in job* of G_i in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t_{cin})$. The computation of $\texttt{CR}_i(t_{cin})$ is presented below (after Eq. (8)).

The maximum total work of the body jobs in the problem window is $(\lfloor \frac{\mathcal{X}_i(t)}{T_i} \rfloor \cdot W_i)$ where $\lfloor \frac{\mathcal{X}_i(t)}{T_i} \rfloor$ is the number of body jobs in the problem window. The maximum total work of the carry-out job during the problem window is at most W_i. Therefore, the inter-task interfering workload of task $G_i \in \texttt{hpt}_k$ is $(\texttt{CR}_i(t_{cin}) + \lfloor \frac{\mathcal{X}_i(t)}{T_i} \rfloor \cdot W_i + W_i)$. The inter-task interfering workload $\mathcal{W}_{k,t}^{inter}$ of all the tasks in $\texttt{hpt}_k$ within the problem window of $v_{k,j}$ is

$$\mathcal{W}_{k,t}^{inter} = \sum_{G_i \in \texttt{hpt}_k} \left(\texttt{CR}_i(t_{cin}) + \lfloor \frac{\mathcal{X}_i(t)}{T_i} \rfloor \cdot W_i + W_i \right) \quad (8)$$

Computing $\texttt{CR}_i(t_{cin})$

We determine the value of $\texttt{CR}_i(t_{cin})$ by considering the work done by the subtasks of the carry-in job of task $G_i \in \texttt{hpt}_k$ in the interval $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t_{cin})$. Consider a subtask $v_{i,g} \in V_i$ of the carry-in job of G_i. Note that R_i^g is the response time of the subtask $v_{i,g} \in V_i$, which is already computed before the response time of $v_{k,j}$ is computed (recall that the response time of all the subtasks of the higher priority task G_i are computed before the response time of $v_{k,j}$ is computed).

Subtask $v_{i,g}$ of task G_i executes in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t_{cin})$ only if its response time R_i^g is $> \texttt{rdy}_k^j$. Based on the similar analysis in Step 2, the higher-priority subtask $v_{i,g}$ of the carry-in job of G_i can execute at most $min\{C_{i,g}, max\{0, R_i^g - \texttt{rdy}_k^j\}\}$ time units inside the problem window of $v_{k,j}$. An upper bound on the total interfering workload of all the higher-priority subtasks in set V_i of the carry-in job of task G_i, denoted by $\mathcal{A}_i$, is

$$\mathcal{A}_i = \sum_{v_{i,g} \in V_i} min\{C_{i,g}, max\{0, R_i^g - \texttt{rdy}_k^j\}\} \quad (9)$$

However, the inter-task interfering workload of the carry-in job of G_i in $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t_{cin})$ cannot be larger than $(m \cdot t_{cin})$ since at most $(m \cdot t_{cin})$ amount of work can be completed within an interval of length t_{cin} on m cores. Therefore, the value of $\texttt{CR}_i(t_{cin})$ is given as follows:

$$\texttt{CR}_i(t_{cin}) = min\{m \cdot t_{cin}, \mathcal{A}_i\} \quad (10)$$

Step 4 (Interference and the Response Time). The total interfering workload, denoted by $\mathcal{I}_{k,j}(t)$, within the problem window of length t for subtask $v_{k,j}$ is the sum of intra-task and inter-task interfering workloads:

$$\mathcal{I}_{k,j}(t) = \mathcal{W}_k^{intra} + \mathcal{W}_{k,t}^{inter} \quad (11)$$

where the values of $\mathcal{W}_k^{intra}$ and $\mathcal{W}_{k,t}^{inter}$ are computed using Eq. (4) and Eq. (8), respectively. The total interfering workload $\mathcal{I}_{k,j}(t)$ causes maximum interference on $v_{k,j}$ in its problem window if all the m cores are simultaneously busy executing the higher-priority interfering workload $\mathcal{I}_{k,j}(t)$. In other words, the execution of subtask $v_{i,j}$ is interfered, after it becomes ready for execution, by at most $\frac{\mathcal{I}_{k,j}(t)}{m}$ time units within its problem window $[\texttt{rdy}_k^j, \texttt{rdy}_k^j + t)$. The response time of subtask $v_{k,j}$ is given using the following recurrence:

$$t^{h+1} \leftarrow \texttt{rdy}_k^j + \frac{\mathcal{I}_{k,j}(t^h)}{m} + C_{k,j} \quad (12)$$

where the first term on the right-hand side in Eq. (12) is the latest time when subtask $v_{k,j}$ becomes ready for execution, the second term represents the interference on subtask $v_{k,j}$ within a problem window of size t^h, and finally the third term $C_{k,j}$ is the WCET of subtask $v_{k,j}$.

The recurrence in Eq. (12) can be solved by searching iteratively the least fixed point that satisfies Eq. (12) starting with $t^0 = C_{k,j}$ for the right-hand side of Eq. (12). This recursion stops when either (i) $t^{h+1} = t^h$ (i.e., subtask $v_{k,j}$ completes at or before the deadline of the task G_k) and we have $R_k^j = t^{h+1}$; or (ii) $t^{h+1} > D_k$ (i.e., task G_k can not guaranteed to be schedulable). If $t^{h+1} > D_k$, we set $R_k^j = \infty$ to specify that the subtask $v_{k,j}$ is not guaranteed to meet the deadline of the task G_k. Given the response time R_k^j of each subtask $v_{k,j}$ of task G_k, the response time of task G_k can be be computed based on Eq. (1). Now we prove in Theorem 1 that the recurrence to compute the response time of each task using Eq. (12) is correct.

Theorem 1. *All the tasks in set $\Gamma = \{G_1, G_2, \ldots G_n\}$ meet their deadlines if $R_k \le D_k$ for $k = 1, 2, \ldots n$.*

Proof. If $R_k \le D_k$, then we also have that $R_k^j \le D_k$ for each subtask $v_{k,j} \in V_k$ from Eq. (1). We prove this theorem by showing that Eq. (12) correctly computes an upper bound on the response time of each subtask $v_{k,j} \in V_k$ of task G_k for $k = 1, 2 \ldots n$.

Based on Eq. (11), we have $\mathcal{I}_{k,j}(t^h) = \mathcal{W}_k^{intra} + \mathcal{W}_{k,t^h}^{inter}$ and the recurrence in Eq. (12) can be re-written as

$$t^{h+1} \leftarrow \texttt{rdy}_k^j + \frac{\mathcal{W}_k^{intra}}{m} + C_{k,j} + \frac{\mathcal{W}_{k,t^h}^{inter}}{m} \quad (13)$$

The intra- and inter-task interfering workloads $\mathcal{W}_k^{intra}$ and $\mathcal{W}_{k,t}^{inter}$ in Eq. (4) and Eq. (8) are computed based on the WCETs of the subtasks in set $\texttt{hpst}_k^j$ and the subtasks of the tasks in set $\texttt{hpt}_k$, respectively.

Lemma 1 proves that the release pattern of the jobs of the higher priority task $G_i \in \texttt{hpt}_k$ according to Figure 3 maximizes the inter-task interfering workload $\mathcal{W}_{k,t}^{inter}$. The maximum inter-task interference is $\mathcal{W}_{k,t}^{inter}/m$ in a problem window of size t.

The intra-task interfering workload $\mathcal{W}_k^{intra}$ is computed by considering the maximum work done by each higher-priority subtask in $\texttt{hpst}_k^j$ that are eligible to execute after time $\texttt{rdy}_k^j$, where $\texttt{rdy}_k^j$ is the latest time when the node $v_{k,j}$ becomes ready for execution.

Node $v_{k,j}$ does not start its execution until it is ready. Apart from the inter-task interference, to complete $C_{k,j}$ amount of execution of node $v_{k,j}$ it may also suffer intra-task interference in the problem window. According to Eq. (13), the execution of node $v_{k,j}$ may be delayed apart from the inter-task interference $\mathcal{W}_{k,t}^{inter}/m$ by at most $\texttt{rdy}_k^j + \frac{\mathcal{W}_k^{intra}}{m}$ time units. Note that $\texttt{rdy}_k^j$ is the latest time by which node $v_{k,j}$ must be ready for execution. If the ancestors of $v_{k,j}$ during actual execution complete earlier than $\texttt{rdy}_k^j$, then the higher-priority non-ancestor subtasks in set $\texttt{hpst}_k^j$ may execute longer in the problem window in comparison to the case when $v_{k,j}$ becomes ready for execution at time $\texttt{rdy}_k^j$. Therefore, the intra-task interference may be lager than $\frac{\mathcal{W}_k^{intra}}{m}$. We will now show that such increase in intra-task interference does not invalidate Eq. (13). This is because if node $v_{k,j}$ becomes ready for execution Δ time units earlier than $\texttt{rdy}_k^j$ during actual execution, then the intra-task interference within the window $[\texttt{rdy}_k^j - \Delta, \texttt{rdy}_k^j)$ during which all the m processors are busy cannot be larger than Δ. And, the intra-task interference beyond time $\texttt{rdy}_k^j$ is at most $\mathcal{W}_k^{intra}/m$. Therefore, the maximum delay in completing the execution of $v_{k,j}$ is at most the sum of (i) the ready time ($\texttt{rdy}_k^j - \Delta$), (ii) the intra-task interference $(\Delta + \mathcal{W}_k^{intra}/m)$, (iii) its own execution time $C_{k,j}$, and (iv) the inter-task interference $\mathcal{W}_{k,t^h}^{inter}/m$ within a problem window of length t_h. The recurrence in such case to determine the response time is

$$t^{h+1} \leftarrow (\mathtt{rdy}_k^j - \Delta) + \Delta + \frac{\mathcal{W}_k^{intra}}{m} + C_{k,j} + \frac{\mathcal{W}_{k,t^h}^{inter}}{m}$$

which is equivalent to the recurrence in Eq. (13), which in turn is equivalent to Eq. (12).

In summary, the cumulative length of the intervals during which $v_{k,j}$ does not execute (either because it is not ready or suffers interference) cannot be larger than $\mathtt{rdy}_k^j + \frac{\mathcal{I}_{k,j}(t^h)}{m}$ where $\mathcal{I}_{k,j}(t^h)$ is the total interfering workload in the problem window of $v_{k,j}$ of length t^h in Eq. (12). Therefore, the recurrence derived in Eq. (12) correctly computes the response time of subtask $v_{k,j}$ by considering the maximum interference that the subtask $v_{k,j}$ in a problem window of length t^h suffers. □

5 Comparison with the State-of-the-art test and an Example

Melani et al. [16] recently proposed a response-time analysis of recurrent DAG tasks scheduled using GFP algorithm. In this section, we present their proposed schedulability test and show that our test is theoretically more precise than the test in [16]. In Section 6, we show that our proposed test empirically performs much better than the test in [16] for scheduling randomly generated DAG task sets.

In [16], each DAG task has a fixed priority but the subtasks of a DAG task G_k are *not* assigned any priority: the subtask-level scheduler is assumed to be an arbitrary greedy scheduler that never idles a core if there is a subtask awaiting execution in the ready queue. In contrast, we consider specific fixed priorities for the subtasks in addition to the fixed priority of each task $G_k \in \Gamma$ to determine which subtask of the highest-priority ready DAG task to execute. In other words, we consider a particular (fixed-priority-based) subtask-level scheduler which is also greedy but selects subtasks for execution in decreasing priority order.

Similar to the analysis proposed in this paper, the response time analysis of each task $G_k \in \Gamma$ in [16] is also performed in a problem window $[r_k, r_k + t)$ of length t where an arbitrary job of task G_k is released at time r_k. The response time of task G_k is computed by determining an upper bound on both intra- and inter-task interference that the *longest path* (i.e., that has length L_k) of G_k suffers in $[r_k, r_k + t)$. In contrast, we determine the response time of G_k by determining an upper bound on both intra- and inter-task interference that each subtask $v_{k,j} \in V_k$ of G_k suffers in $[r_k, r_k + t)$.

Next we present the response time test that is proposed by Melani et al. in [16] and then we show that our proposed test is theoretically better than the state-of-the-art test.

According to the work in [16], the intra-task interference on the longest path of an (arbitrary) job of G_k is $\frac{W_k - L_k}{m}$ where W_k and L_k are the total work of all the nodes and the length of the longest path in G_k. For computing the inter-task interference, the worst-case release pattern that maximizes the workload in the problem window (given in Figure 4) is derived in [16] by considering that (i) the carry-in job of $G_i \in \mathtt{hpt}_k$ starts execution at time r_k and finishes at time $(r_k + W_i/m)$ such that $(r_k + W_i/m - R_i)$ is the release time of the carry-in job, (ii) each of the later jobs executes W_i amount of work in W_i/m time units as soon as possible. An upper bound on the inter-task interfering workload (denoted as $\overline{\mathcal{W}}_{k,t}^{inter}$) of all the higher-priority tasks in $\mathtt{hpt}_k$ in a problem window of length t is given as follows [16]:

$$\overline{\mathcal{W}}_{k,t}^{inter} = \sum_{G_i \in \mathtt{hpt}_k} \Bigg(\left\lfloor \frac{t + R_i - W_i/m}{T_i} \right\rfloor W_i + min\left\{ W_i, m \cdot (t + R_i - \frac{W_i}{m}) \ mod\ T_i \right\} \Bigg) \quad (14)$$

The total inter-task interference on the longest path of G_k in the problem window is $(\sum_{G_i \in \mathtt{hpt}_k} \overline{\mathcal{W}}_{i,t}^{inter})/m$ and the intra-task interference on the longest path of G_k is $(W_k - L_k)/m$. The response-time of task G_k for GFP scheduling is given in [16] using the following recurrence:

$$t^{h+1} \leftarrow L_k + \frac{W_k - L_k}{m} + \frac{\overline{\mathcal{W}}_{k,t^h}^{inter}}{m} \quad (15)$$

Similar to the recurrence in Eq. (12), the recurrence in Eq. (15) is solved by searching the least fixed point starting with $t^0 = L_k$ from the right-hand side.

Determining the workload in the problem window plays one of the most important roles in gauging the effectiveness of the state-of-the-art test in Eq. (15). The analysis in [16] hides the internal structures of the DAG task G_i by using only two parameters to represent this task: W_i and L_i. Such abstraction while being simple to analyze leads to computing the workload in the problem window more pessimistically, which may lead task sets to be deemed unschedulable using the proposed test in Eq. (15) while the task set is in fact schedulable. Our proposed technique to compute the workload is less pessimistic (will be shown shortly in Theorem 2) due to exploiting the precedence constraint information (i.e., subtask-level priority) of the tasks.

It will shown in Theorem 2 that our proposed response time test in Eq. (12) dominates the test in Eq. (15) in the sense that: if a task set is schedulable using the test in Eq. (15), then that task set is also guaranteed to be schedulable using our proposed test in Eq. (12); and there are task sets (see Example 1) for which our test guarantees schedulability but the test in Eq. (15) cannot guarantee schedulability.

Theorem 2. *The response-time test in Eq.* (12) *dominates the response-time test in Eq.* (15).

Proof. The response time tests in Eq. (12) and Eq. (15) for task $G_k \in \Gamma$ can be rewritten as the sum of the following two factors F1 and F2:

- F1. Response time of task G_k without considering the inter-task interference, and
- F2. Inter-task interference in a problem window of length t.

Based on Eq. (11), we re-write Eq. (12) as follows.

$$t^{h+1} \leftarrow \mathtt{rdy}_k^j + \frac{\mathcal{W}_k^{intra}}{m} + C_{k,j} + \frac{\mathcal{W}_{k,t_h}^{inter}}{m} \quad (16)$$

where

- F1: $\mathtt{rdy}_k^j + \frac{\mathcal{W}_k^{intra}}{m} + C_{k,j}$ is the response time of task G_k without considering the inter-task interference; and
- F2: $\frac{\mathcal{W}_{k,t_h}^{inter}}{m}$ is the inter-task interference in a problem window of length t_h.

Similarly, in Eq. (15), we have

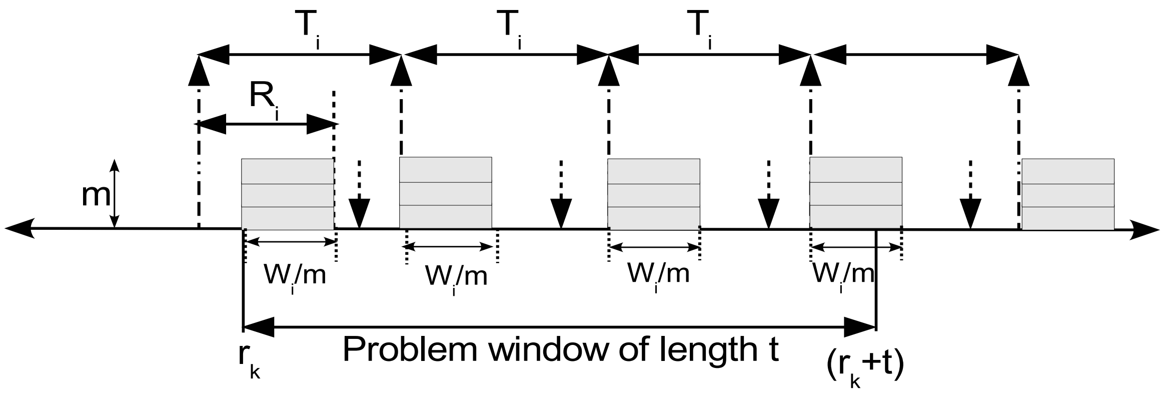


Figure 4: Worst-case release pattern for workload computation in [16].

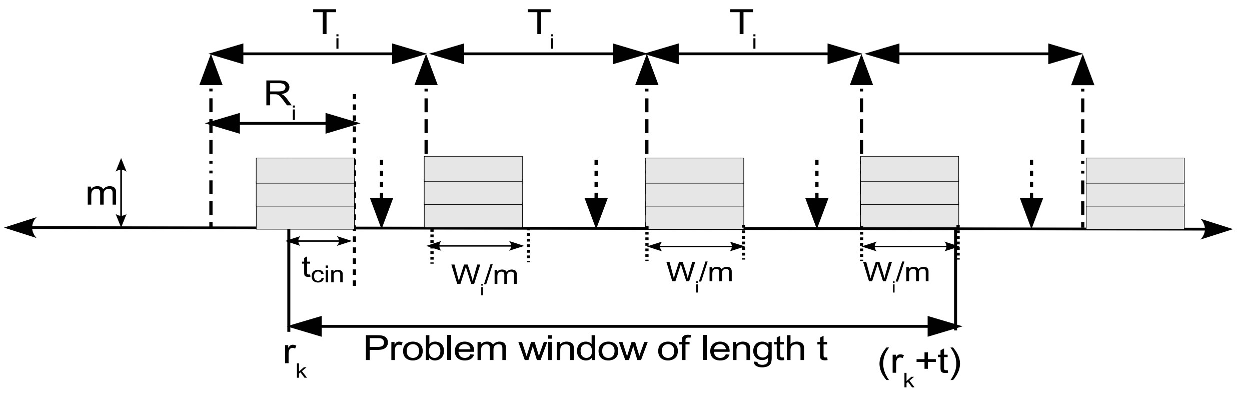


Figure 5: Left Shift of the worst-case release pattern for workload computation in [16].

- F1: the term $L_k + \frac{W_k - L_k}{m}$ is the response time of task G_k without considering the inter-task interference; and
- F2: $\frac{\overline{\mathcal{W}}^{inter}_{k,t_h}}{m}$ is the inter-task interference in a problem window of length t_h.

To prove this theorem, we first show that each of the terms F1 and F2 is never more pessimistic in Eq. (12) in comparison to the state-of-the-art test test in Eq. (15).

Factor F1. The quantity $L_k + (W_k - L_k)/m$ in Eq. (15) is the response time of DAG G_k under any work-conserving (i.e., greedy) algorithm without considering the inter-task interference. In contrast, we assign fixed priorities to the subtasks and use a fixed-priority based subtask level scheduler to schedule the tasks. Since fixed-priority scheduler is also a work-conserving (i.e., greedy) scheduler, the response time of DAG task G_k, i.e., value of $\texttt{rdy}^j_k + \frac{\mathcal{W}^{intra}_k}{m} + C_{k,j}$, is not larger than $L_k + (W_k - L_k)/m$. Therefore, factor F1 in our test is never more pessimistic than the factor F1 in the state-of-the-art test.

Factor F2. We will show that $\overline{\mathcal{W}}^{inter}_{k,t} \geq \mathcal{W}^{inter}_{k,t}$ for any problem window of size t, which implies that the inter-task interfering workload computed using our approach is less than or equal to that of computed by Melani et al. in [16] for the same length of the problem window.

First, we show another worst-case release pattern (Figure 5) that is *equivalent* to the release pattern in Figure 4 considered by Melani et al [16]. The inter-task interfering workload $\overline{\mathcal{W}}^{inter}_{i,t}$ in Eq. (15) is exactly equal to the inter-task interfering workload of this equivalent release pattern in Figure 5.

Second, we show that the release pattern in Figure 5 computes the inter-task interfering workload more pessimistically in comparison to the release pattern that we consider in Figure 3. Since the release patterns in Figure 4 and Figure 5 are equivalent, we conclude that Figure 4 (the state-of-the-art) computes the inter-task interfering workload more pessimistically in comparison to the release pattern that we consider in Figure 3. Therefore, $\overline{\mathcal{W}}^{inter}_{k,t} \geq \mathcal{W}^{inter}_{k,t}$.

Consider shifting the problem window in Figure 4 to the right such that the body job finishes its execution of W_i time units in an interval of length W_i/m exactly at the end of the window, i.e., at time $(r_k + t)$. This shifted window is shown in Figure 5. It is evident that the inter-task interfering workload in the problem window in Figure 5 is same as that of in Figure 4. This is because the amount of work that is decreased from the left-hand side due to the right shift is exactly equal to the amount of work that is increased from the right-hand side of the problem window. In other words, the inter-task interfering workload for the release pattern in Figure 5 is also $\overline{\mathcal{W}}_{k,t}^{inter}$.

The release pattern in Figure 5 is quite similar to the release pattern in Figure 3 except that within the window of length $[r_k, r_k + t_{cin})$, the work done by the carry-in job of the higher-priority task G_i is exactly $(m \cdot t_{cin})$ in Figure 5 while it may be smaller than $(m \cdot t_{cin})$ in Figure 3. In other words, while the work done by the body and the carry-out jobs of G_i is same in our release pattern (Figure 3) and the equivalent release pattern (Figure 5), our approach to compute the work of the carry-in job of a higher priority task G_i is less pessimistic. This is due to the *min* function in Eq. (10): if $\texttt{CR}_i(t_{cin}) = (m \cdot t_{cin})$ in Eq. (10), then the carry-in workload in $[r_k, r_k + t_{cin})$ computed using Eq. (10) is same as that of computed in [16]. If $\texttt{CR}_i(t_{cin}) < (m \cdot t_{cin})$, then the carry-in workload computed using Eq. (10) in $[r_k, r_k + t_{cin})$ is strictly smaller than that of computed in [16]. Consequently, the proposed technique to compute the work of the carry-in job of G_i in $[r_k, r_k + t_{cin})$ is smaller than or equal to that of in Figure 5.

All the body jobs and the carry-out job of the higher priority task G_i execute exactly for W_i time units in both Figure 3 (our analysis) and Figure 5. Since the inter-task interfering workload in Figure 5 is $\overline{\mathcal{W}}_{i,t}^{inter}$, our proposed technique to compute the carry-in workload in $[r_k, r_k + t)$ is smaller than or equal to that of proposed in [16], i.e., $\overline{\mathcal{W}}_{i,t}^{inter} \geq \mathcal{W}_{i,t}^{inter}$ for any problem window of length t. Consequently, the inter-task interference using our approach is smaller than or equal to that of computed in [16].

Since each of the factors F1 and F2 in our approach is never worse than the state-of-the-art in [16], the computed response time in Eq. (1) is never larger than the response time of the state-of-the-art in [16]. Example 1 shows that the response time computation in Eq. (1) is strictly smaller than that of computed using Eq. (15) for an example DAG task in Figure 6, which show the dominance of the proposed test over the state-of-the-art test. □

Next we show an example of a single DAG task where the response time computed by Eq. (15) is larger than that of computed using our approach in Eq. (12). When applied to a single DAG task G_k (i.e., there is no inter-task interference), the response time of G_k based on Eq. (15) is

$$R_k = L_k + \frac{W_k - L_k}{m} \tag{17}$$

The response time of each subtask $v_{k,j}$ of G_k based on Eq. (12) is given as follows:

$$R_k^j = \texttt{rdy}_k^j + \frac{\mathcal{W}_k^{intra}}{m} + C_{k,j} \tag{18}$$

Example 1. *Consider* GFP *scheduling of a single DAG task G_k in Figure 6 on $m = 2$ cores where $D_k = 52$ and $T_k = 100$. The length of the longest path is $L_k = 46$ and total work of G_k is $W_k = 64$. The response time using Eq. (17) is $L_k + (W_k - L_k)/m = 46 + (64 - 46)/2 = 55$. Since $D_k = 52 < 55$, the state-of-the-art test cannot guarantee schedulability of G_k.*

Now we compute the response time of G_k using Eq. (18). The fixed priorities of the subtasks of G_k are $v_{k,1} \succ v_{k,3} \succ v_{k,2} \succ v_{k,5} \succ v_{k,4} \succ v_{k,6}$. The response time of each subtask based on Eq. (18) is given in the last column in Table 1. The sink subtask $v_{k,6}$ has response time $R_k^6 = 50.5$. Therefore, $R_k = R_k^6 = 50.5$ based on Eq. (1). Since $D_k = 52 \geq R_k = 50.5$, we can guarantee that the task G_k meets its deadline using the proposed test.

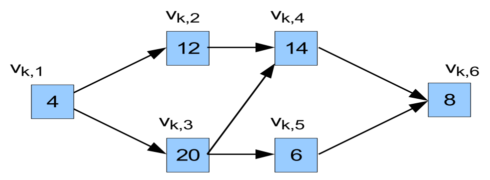


Figure 6: An example DAG task.

6 Empirical Study

In this section, we compare the performance of our proposed schedulability test with the state-of-the-art schedulability test using randomly-generated parallel DAG task sets. The tasks are assigned fixed priorities based on deadline-monotonic (DM) priority ordering. We denote our proposed schedulability test in Eq. (12) by "Our-DM" and the state-of-the-art schedulability tests in Eq. (15) by "MBBMB-DM". Note that unlike MBBMB-DM test, Our-DM test considers subtask-level fixed priority (i.e., topological sort in Section 2).

The utilization u_k of a task G_k is the ratio between its total work and the period, i.e., $u_k = W_k/T_k$. The utilization u_k of the DAG in Figure 6 is $W_k/T_k = 64/100 = 0.64$. For a task set Γ, its total utilization is defined as $U = \sum_{G_k \in \Gamma} u_k$. The utilization U of a task set specifies its computation load. Before presenting the experimental results, the task sets generation algorithm is presented.

6.1 Task Sets Generation Algorithm

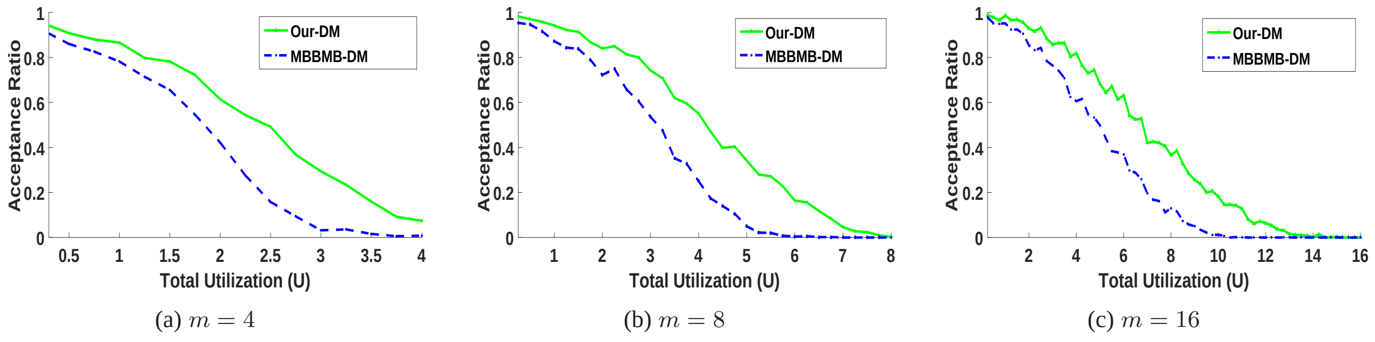
We use a similar DAG task set generation algorithm that is used in [16] which generates a series of parallel graphs by recursively expanding the *non-terminal* vertices for a given recursion depth. The purpose of such expansion is to generate either a new *terminal* vertices (i.e., sink subtasks) or a new *parallel subgraphs*. The maximum recursion depth for all our experiments is set to 2.

The probabilities to generate a terminal vertex and a parallel subgraph by expanding a non-terminal vertex are p_{term} and $(1 - p_{term})$, respectively. The number of branches of a parallel subgraph is uniformly selected from $[2, n_{par}]$ where n_{par} is the maximum branches (degree) for any parallel subgraph. Random edges are added between pairs of subtasks with probability p_{add} in a way so that there is no cycle in the graph. Once the subtasks and the edges of a DAG task G_k are generated, the parameters of each DAG task G_k are generated as follows:

- the WCET $C_{k,i}$ of a subtask $v_{k,i} \in V_k$ of task G_k is uniformly selected in the range $[1, 100]$;
- the length of the longest path L_k and the workload W_k are computed;
- the period T_k (i.e., the minimum inter-arrival time) is uniformly selected in the interval $[L_k, W_k/\beta)$, where $\beta \leq 1$ is used to control the minimum utilization of task G_k. The utilization u_k of G_k is in $[\beta, W_k/L_k]$; finally,

subtask	$C_{k,j}$	$Ancst_k^j$	$\mathtt{rdy}_k^j = \max\limits_{v_{k,a} \in Ancst_k^j} R_k^a$	$\mathcal{S}_k^j = \mathtt{hpst}_k^j - Ancst_k^j$	$\mathcal{W}_k^{intra}$	$R_k^j = \mathtt{rdy}_k^j + \frac{\mathcal{W}_k^{intra}}{m} + C_{k,j}$
$v_{k,1}$	4	$\emptyset$	0	$\emptyset$	0	4
$v_{k,3}$	20	$v_{k,1}$	4	$\emptyset$	0	4+0+20=24
$v_{k,2}$	12	$v_{k,1}$	4	$\{v_{k,1}, v_{k,3}\} - \{v_{k,1}\} = \{v_{k,3}\}$	20	$4 + \frac{20}{2} + 12 = 26$
$v_{k,5}$	6	$v_{k,1}, v_{k,3}$	24	$\{v_{k,2}\}$	26-24=2	$24 + \frac{2}{2} + 6 = 31$
$v_{k,4}$	14	$v_{k,1}, v_{k,2}, v_{k,3}$	26	$\{v_{k,5}\}$	31-26=5	$26 + \frac{5}{2} + 14 = 42.5$
$v_{k,6}$	8	$v_{k,1}, \ldots, v_{k,5}$	42.5	$\emptyset$	0	$42.5 + 0 + 8 = 50.5$

Table 1: Computing response time of the DAG G_k in Figure 6 where the priorities of the subtasks are $v_{k,1} \succ v_{k,3} \succ v_{k,2} \succ v_{k,5} \succ v_{k,4} \succ v_{k,6}$. From Eq. (4), we have $\mathcal{W}_k^{intra} = \sum_{v_{k,h} \in \mathcal{S}_k^j} min\{C_{k,h}, max\{0, R_k^h - \mathtt{rdy}_k^j\}\}$.



(a) $m = 4$ (b) $m = 8$ (c) $m = 16$

Figure 7: Acceptance ratio of Our-DM and MBBMB-DM tests for $m = 4, 8, 16$.

- the relative deadline D_k of G_k is uniformly selected from the range $[L_k, T_k]$.

6.2 Experimental Results

We set $p_{term} = 0.5$, $n_{par} = 5$, $p_{add} = 0.1$ and $\beta = 0.1$ for our experiments. For a target total utilization U of a task set, new DAG tasks are repeatedly added to the task set until the total utilization of the task set is equal to U. In order to generate a task set with total utilization exactly equal to U, the period of the last task is adjusted so that the total utilization of the task set is exactly equal to U. A total of 500 task sets are generated at each of the utilization levels $U \in [0.25, 0.5, \ldots m]$ where m is the number of cores. Each of the 500 task sets that are generated at that particular utilization level $U \in [0.25, 0.5, \ldots m]$ has total utilization U.

For a specific schedulability test, and m, U values, we consider the metric called *acceptance ratio* that denotes the fraction of task sets out of 500 DAG task sets that are guaranteed to be schedulable by the schedulability test at that utilization level on m cores. We implemented our experiments in MATLAB and computed the acceptance ratios of both tests for various values of U and m.

Figure 7 presents the acceptance ratios of Our-DM test and MBBMB-DM test for $m = 4, 8, 16$ cores. The x-axis in each plot in Figure 7 represents the total utilization U of each task set and the y-axis is the acceptance ratio. Our proposed Our-DM schedulability test outperforms the state-of-the-art MBBMB-DM test. For example, the acceptance ratios in Figure 7a at $U = 2.0$ for $m = 4$ of Our-DM test and MBBMB-DM test are respectively around 50% and 70%. In other words, around 20% more task sets are deemed to be schedulable using Our-DM test in comparison to MBBMB test at $U = 2.0$ and $m = 4$. The difference in acceptance ratios between Our-DM and MBBMB-DM tests increases with the increase in total utilization of the task sets for all $m = 4, 8, 16$ in Figure 7a–7c. In Figure 7b for $m = 8$, around 20% and 30% more task sets are deemed to be schedulable respectively at $U = 3.0$ and $U = 5.0$ using Our-DM test in comparison to MBBMB test. It is generally more difficult to schedule task sets with relative higher total utilization or higher load. Our-DM test is very effective in comparison to MBBMB-DM test in guaranteeing schedulability of relatively high-utilization task sets.

Unlike the response time computation using MBBMB-DM test, the schedulability analysis used to derive Our-DM test uses the internal structure of each DAG task to determine which subtasks may execute in parallel with the execution of the subtasks in the longest path of the DAG. Consequently, the response time computation using Our-DM test is more precise than that of MBBMM-DM test.

Variation with number of cores (m). For this set of experiments, we vary the number of cores while keeping the total utilization U of each task set fixed. Figure 8 shows the acceptance ratio of Our-DM test and MBBMB-DM test for various values of m using three different (fixed) total utilizations: $U = 2$, $U = 4$, and $U = 8$. When the number of cores becomes larger, the acceptance ratio also increases in all the three plots in Figure 8a–8c. This is expected because a relative higher number of cores has higher likelihood of meeting all the deadline of the task sets for both tests.

However, for many resource-constrained systems where the size, weight, and power are limited and costly, we cannot afford a much larger number of cores. In such case (i.e., when the number of cores is small), Our-DM test has much larger acceptance ratio than that of the MBBMB-DM test. For example in Figure 8a where $U = 2.0$, Our-DM test has 80% acceptance ratio when $m \geq 8$ while the MBBMB-DM test reaches 80% acceptance when $m \geq 20$ cores. In Figure 8c where $U = 8.0$, Our-DM test has larger than 80% acceptance ratio when $m \geq 64$ while the MBBMB-DM

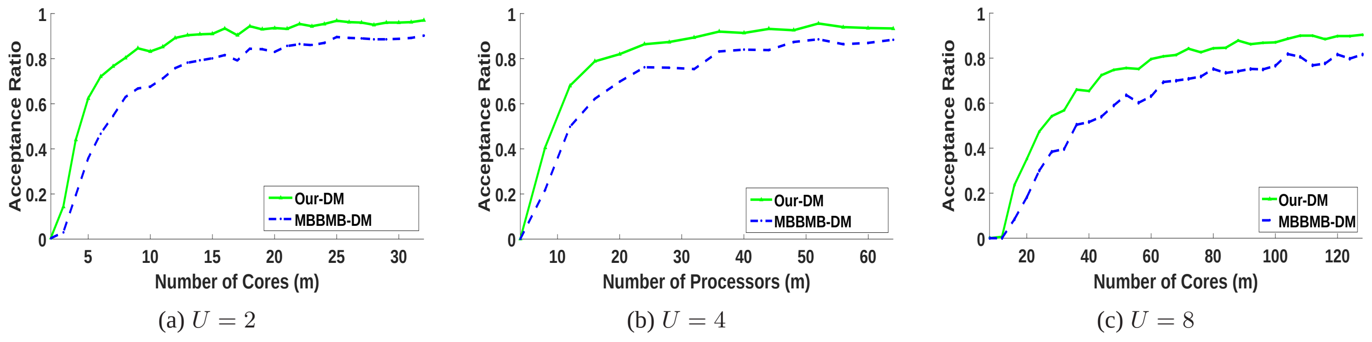


Figure 8: Acceptance ratios of Our-DM and MBBMB-DM tests for varying number of cores.

test has 80% acceptance when $m \geq 100$ cores. These results show the effectiveness of our proposed schedulability test in reducing the demand for hardware resources (particularly, the number of cores), which in turn reduces the cost of developing the systems for mass production.

Variation with number of tasks (n). We vary the number of tasks while keeping the total utilization (U) of each task set and number of cores (m) fixed. Since the number of tasks (n) of each task set is fixed for a fixed value of U, the total utilization of n tasks in each randomly-generated task set must be equal to the target utilization U. The `UUnifast` algorithm, proposed by Bini and Buttazzo [23], with parameters n and U is used to generate the utilization values of the n tasks in a task set such that the total utilization of the task set is exactly U. Figure 9 shows the acceptance ratios for various values of n for two given configurations of (U, m): Figure 9a–9b shows results for $(U = 2, m = 4)$ and $(U = 4, m = 8)$, respectively.

When the number of tasks (n) increases while the total utilization U is unchanged, the number of low-utilization tasks in a task set increases because the total utilization U is distributed across a relatively larger number of tasks of the task set. A low-utilization task uses less computing resource and provides more opportunity for other tasks to execute on the cores (i.e., easier bin-packing). Consequently, when the number of tasks in the random tasks sets is relatively larger (e.g., $n \geq 6$ in Figure 9a and $n \geq 14$ in in Figure 9b), then the acceptance ratios of both tests are 100%.

When the number of tasks is relatively small (i.e., there are more high-utilization tasks in a task set), then Our-DM test has higher acceptance ratio than that of the MBBMB-DM test. For example, when $U = 4$ and $m = 8$ in Figure 9b, the acceptance ratio for $n = 5$ of Our-DM test is $\approx 65\%$ while it is $\approx 40\%$ for MBBMB-DM test. Therefore, our test is more effective in scheduling task sets having a relatively larger number of high-utilization tasks in comparison to task sets with higher number of low-utilization tasks for the same total utilization U.

6.3 Other tests: Implicit Deadlines

Some recent works [24], [25] have proposed schedulability tests for DAG tasks where the relative deadline D_i is equal to the period T_i for each task G_i (known as implicit-deadline task sets) by considering scheduling policies, like federated scheduling, decomposition-based scheduling. This subsection presents the acceptance ratios of the following four schedulability tests (in addition to MBBMB-DM and Our-DM) for implicit deadline tasks:

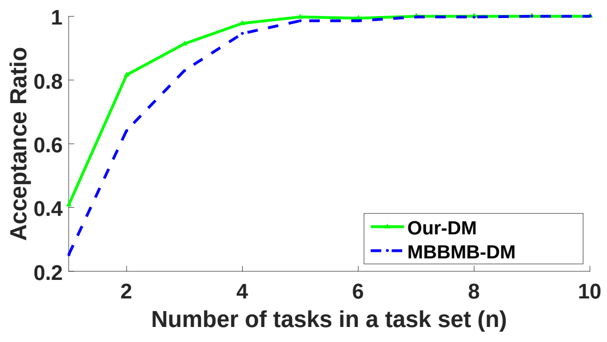
- MBBMB-EDF: This test in [16] considers global EDF scheduling of DAG tasks.
- DECOM-EDF: This test in [25] considers decomposing each DAG task by assigning artificial release time and artificial deadline to each subtask where such subtasks are scheduled using global EDF policy.
- FED-LI: This test in [24] considers federated scheduling where a DAG task with high utilization is assigned a dedicated number of processors while all the light-utilization tasks are executed sequentially and scheduled on a shared set of processors.
- SIM:For each set of DAG tasks, their execution is simulated using our proposed two-level `GFP` scheduler where the release time of each DAG task is zero and the jobs of each task are released strictly periodically. The simulation of the schedule is run for 10^6 time units. The acceptance ratio of SIM is an upper bound on the acceptance ratio the random task sets that are actually schedulable using the proposed scheduler. The acceptance ratio of SIM depicts the tightness of our proposed schedulability test MBBMB-DM with respect to a hypothetical exact test.

The acceptance ratios for SIM, Our-DM, MBBMB-DM, FED-LI, DECOM-EDF and MBBMB-EDF tests are shown in Figure 10 for $m = 4$ and $m = 8$ processors. It can be observed that our proposed Our-DM performs better than all other tests. More importantly, the difference between the plot of Our-DM is not very far from the plot for SIM. This signifies that the preciseness of Our-DM test is close to the (hypothetical) exact test for our proposed two-level GFP scheduling policy.

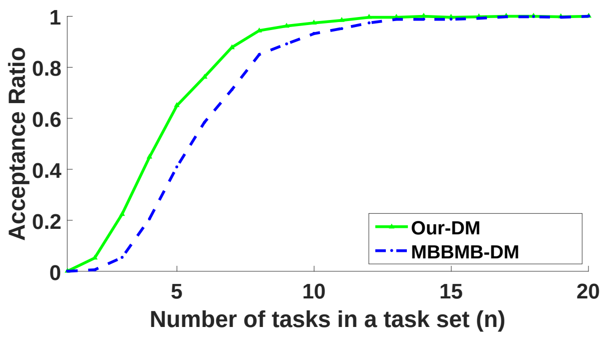
7 Related Works

Many of the earlier works on parallel task models proposed resource-augmentation bounds and schedulability tests for various scheduling algorithms. The resource augmentation bound $\mathcal{B}$ of a scheduling algorithm $\mathcal{A}$ indicates that if there is a way to schedule a task set on m identical unit-speed cores, then algorithm $\mathcal{A}$ is guaranteed to successfully schedule the same task set on m cores with each core being $\mathcal{B}$ times as fast as the original. However, resource-augmentation bound cannot be used as a schedulability test since it is derived based on a hypothetical optimal scheduler.

The works on real-time scheduling of parallel tasks on multicores can be categorized in three groups considering the task model that each group considers: (i) fork-join model [7], (ii)

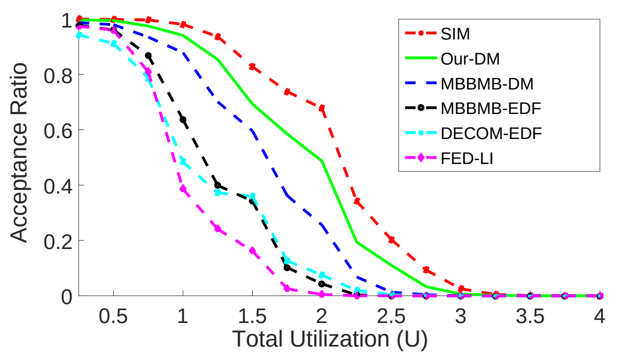


(a) $U = 2$ and $m = 4$



(b) $U = 4$ and $m = 8$

Figure 9: Acceptance ratios by varying n.



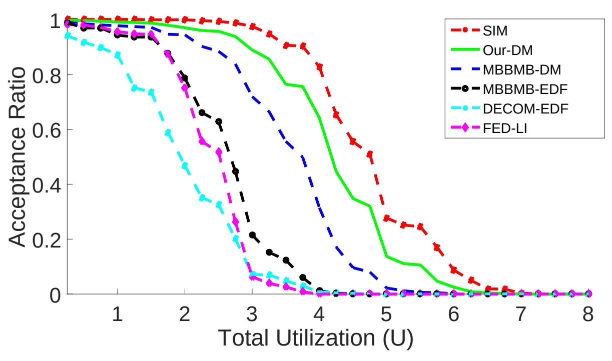


Figure 10: Acceptance ratios by varying U for implicit-deadline tasks for $m = 4$ (left-hand side graph) and for $m = 8$ (right-hand side graph).

synchronous parallel task model [8], [9], [10], (iii) the dag task model [12], [13], [14], [15], [16].

Each task in fork-join model has alternating sequential and parallel segments [7]. The fork-join task model is generalized in synchronous parallel task model by allowing multiple parallel segments to execute without a sequential segment in between [8], [9], [10]. Different resource augmentation bounds and schedulability tests are proposed for both fork-join and synchronous parallel task models [7], [8], [9], [10].

The third group of works considers dag task model which is more general than the synchronous task model [12], [13], [14], [15], [16]. The work in [12] proposed both a polynomial and a pseudo-polynomial schedulability test for scheduling a single sporadic (i.e., recurrent) dag task using global dynamic-priority-based Earliest-Deadline-First (GEDF) scheduling. The works in [13], [14] derived resource augmentation bound of $(2 - 1/m)$ for GEDF scheduling of multiple sporadic dag tasks, where m is the number of unit-speed cores. The work in [13] also derived a resource augmentation bound of $(3 - 1/m)$ for global fixed-priority-based deadline-monotonic scheduling. The work by Melani et al. [16] derived a schedulability test for GFP scheduling. The analysis in [16] is the state-of-the-art result for response-time computation of multiple recurrent DAG tasks. We have shown that the performance of our proposed test is much better than the test in [16].

In addition to global scheduling, researchers have analyzed at least two other mechanisms to schedule parallel DAG tasks: Federated scheduling [24] and Decomposition-based scheduling [25]. The main idea of federated scheduling is the following: each DAG task with utilization larger than 1 (called, high-utilization task) is assigned a dedicated number of processors while each of the other tasks with utilization no larger than 1 (called, low-utilization task) executes sequentially on the remaining processors. It is shown in [24] that federated scheduling has a capacity augmentation bound of 2 for the case when the number of processor is large.

In decomposition based scheduling, each DAG task is transferred into a set of independent sporadic task by inserting artificial release time and artificial deadline for each of the subtasks of the DAG task. The artificial release times and deadlines to each subtask are assigned such that the precedence constraints of the original DAG task are satisfied. The decomposed subtasks of all the DAG tasks are scheduled based on GEDF scheduling policy in [25]. We have shown that our proposed Our-DM test empirically performs better than both the federated scheduling [24] and the decomposition-based scheduling in [25].

There are many works on scheduling parallel tasks on multiprocessors without concerning real-time requirements [3], [20], [21]. The Heterogeneous Earliest Finish Time (HEFT) scheduler in [21] prioritizes the subtasks of the DAG based on the structure of the DAG. The HEFT algorithm sorts the tasks based on the decreasing order of their upward rank. The task with the highest upward rank is dispatched to the processor that the task will have the smallest execution time. Similarly, the Critical-Path-on-a-Processor (CPOP) scheduler, also proposed in [21], in addition to the upward rank takes into account also the downward rank and the tasks are sorted based on this two characteristics of the DAG.

For both HEFT and CPOP schedulers, two subtasks can have the same rank and ties are broken randomly. Such random tie breaking may lead the subtasks to execute differently in different executions. Moreover, if some subtask takes less than its WCET, then the DAG task can generate a relatively longer schedule than the schedule that is generated when each subtask takes its WCET, which is known as execution-time based timing anomalies [19]. There is no built-in strategy in such algorithm to avoid timing anomaly and therefore cannot be applied to real-time systems. In short, scheduling algorithms designed for improving the average case performance, like the HEFT or CPOP algorithms, cannot be directly applied to guarantee real-time constraints.

8 Conclusion

This paper presents a schedulability analysis of recurrent DAG tasks considering GFP scheduling at both subtask and task levels. A simple method to assign fixed priorities to the subtasks of a DAG task is proposed based on the structure (i.e., topology) of each DAG task. To the best of our knowledge, assigning fixed priorities to the subtasks of a DAG task by exploring the internal structure of each DAG task has not been addressed in any earlier work. Based on the priorities of the subtasks, a new technique to compute the response time of each subtasks is presented. Simulation results using randomly generated tasks show that our proposed test performs better than the state-of-the-art test, particularly, for task sets with relatively larger utilization and also for task set with relatively higher number of high-utilization tasks. Finding a more effective priority assignment for the subtasks of each task is left as a future work.

Acknowledgments

This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

References

[1] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Third Edition, Springer*. Third Edition, Springer, 2011.
[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of AFIPS*, 1967.
[3] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.
[4] "Openmp application program interface. version 4.0, jul 2013."
[5] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Parallel Processing, 2009. ICPP'09. International Conference on*. IEEE, 2009, pp. 124–131.
[6] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu, "Wcet analysis of a parallel 3d multigrid solver executed on the merasa multi-core," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
[7] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. of RTSS*, 2010.
[8] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proc. of RTSS*, 2011.
[9] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *Proc. of ECRTS*, July 2012, pp. 321–330.
[10] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for synchronous parallel tasks on multicore platforms," in *Proc. of ECRTS*, 2013.
[11] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proc. of RTNS*, 2014.
[12] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Proc. of RTSS*, 2012.
[13] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *Proc. of ECRTS*, 2013.
[14] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *Proc. of ECRTS*, 2013.
[15] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic dag task systems," in *Proc. of ECRTS*, 2014.
[16] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *Proc. of ECRTS*, 2015.
[17] B. Fechner, U. Honig, J. Keller, and W. Schiffmann, "Fault-tolerant static scheduling for grids," in *Proc. of IPDPS*, 2008.
[18] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are static schedules so bad? a case study on cholesky factorization," in *Proc. of IPDPS*, 2016.
[19] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.
[20] "Multi-objective energy-efficient workflow scheduling using list-based heuristics," *Future Generation Computer Systems*, vol. 36, pp. 221 – 236, 2014.
[21] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, 2002.
[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2001.
[23] E. Bini and G. C. Buttazzo, "Measuring the Performance of Schedulability Tests," *Real-Time Systems*, vol. 30, pp. 129–154, 2005.
[24] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *Proc. of ECRTS*, 2014.
[25] X. Jiang, X. Long, N. Guan, and H. Wan, "On the decomposition-based global edf scheduling of parallel real-time tasks," in *Proc. of RTSS*, 2016.

Risat Pathan is an assistant professor in the Department of Computer Science and Engineering at Chalmers University of Technology, Sweden. He received the M.S., Lic.-Tech., and Ph.D. degrees from Chalmers University of Technology in 2006, 2010, and 2012, respectively. He visited the Real-Time Systems Group at The University He visited the Real-Time Systems Group at The University of North Carolina at Chapel Hill, USA during fall 2011. His main research interests are real-time scheduling on uni- and multi-core processors from efficient resource utilization, fault-tolerance and mixed-criticality perspectives. Risat is a member of the IEEE and the ACM.

Petros Voudouris is a PhD student in the Department of Computer Science and Engineering at Chalmers University of Technology, Sweden. Petros received his B.Sc degree in Computer Science from University of Crete, Heraklion, Greece in 2011. He received his M.Sc. degree in Embedded Systems from department of Computer Science and Mathematics, Technical University of Eindhoven, Netherlands, in 2014. His main research interests are the design and analysis of time-predictable scheduling algorithm and worst-case execution time analysis for parallel computer systems. Petros is a student member of IEEE.

Per Stenström is professor at Chalmers University of Technology. His research interests are in parallel computer architecture. He has authored or co-authored four textbooks, more than 150 publications and ten patents in this area. He has been program chairman of several top-tier IEEE and ACM conferences including IEEE/ACM Symposium on Computer Architecture and acts as Senior Associate Editor of ACM TACO and Associate Editor-in-Chief of JPDC. He is a Fellow of the ACM and the IEEE and a member of Academia Europaea, the Royal Swedish Academy of Engineering Sciences and the Royal Spanish Academy of Engineering Science.