Tardiness Bounds for Sporadic Gang Tasks under Preemptive Global EDF Scheduling

Zheng Dong, *Member, IEEE,* Kecheng Yang, *Member, IEEE,* Nathan Fisher, *Member, IEEE,* Cong Liu, *Member, IEEE,*

Abstract—Following the trend of increasing autonomy in cyber-physical systems, parallel embedded architectures have enabled devices to better handle the large streams of data and intensive computation required by such autonomous systems. However, while the explosion of highly-parallel platforms has seen a proportional growth in the number of applications/devices that utilize these platforms, the embedded systems community's understanding of how to build time-predictable, safety-critical systems with parallel platforms has not kept pace. As a well-motivated but challenging parallel scheduling model, gang scheduling requires all parallel threads of each parallel task to simultaneously execute in unison, which is in contrast to traditional, multi-threaded parallel scheduling, where a parallel task may spawn multiple threads, and each thread will be scheduled independently of other threads of the same task. While increasing research efforts on hard real-time (HRT) gang scheduling have recently been seen, the problem of gang scheduling in the context of soft real-time (SRT) systems, where provably bounded deadline tardiness can be tolerated, has hardly been studied yet. In this paper, we derive and prove the first tardiness bounds for sporadic gang task systems under preemptive GEDF scheduling. A total utilization bound for SRT-schedulability is required for ensuring such tardiness bounds but it is shown to be tight with respect to the platform capacity and maximum parallelism-induced idleness. Furthermore, we also empirically evaluate the effects of different degrees of task parallelism upon the SRT-schedulability.

Index Terms—Real-time scheduling, gang tasks, schedulability test, tardiness bound, Global-Earliest-Deadline.

1 Introduction

A major factor in the recent drive towards increasingly autonomous systems (e.g., autonomous automobiles [16], drones [7], etc.) is the proliferation of relatively inexpensive, yet highly-parallel embedded architectures [12], [11]. These parallel embedded architectures (e.g., graphics processing units [18], tensor processing units [9], etc.) have enabled devices to better handle the large streams of data and intensive computation required to learn and make decisions autonomously in uncertain, high-dimensional environments with techniques like deep learning [13]. However, while the explosion of highly-parallel platforms has seen a proportional growth in the number of applications/devices that utilize these platforms, the embedded systems community's understanding of how to build time-predictable, safety-critical systems with parallel platforms has not kept pace. In fact, the challenges and difficulty in guaranteeing timing constraints in a parallel platform typically are only compounded as architectures increase the number of parallel processing resources on an embedded board; this is due to the increasing potential for contention between competing tasks executing on the different resources (e.g., contention for a shared resource between two threads executing in parallel on different cores).

- *Zheng Dong and Nathan Fisher are with the Department of Computer Science, Wayne State University, Detroit, MI 48202, USA.(e-mail: dong@wayne.edu, dx3281@wayne.edu)*
- *Kecheng Yang is with the Department of Computer Science, Texas State University, San Marcos, TX 78666, USA (e-mail: yangk@txstate.edu).*
- *C. Liu is with the Department of Computer Science, The University of Texas at Dallas, Dallas, TX 75080, USA (e-mail: cong@utdallas.edu).*

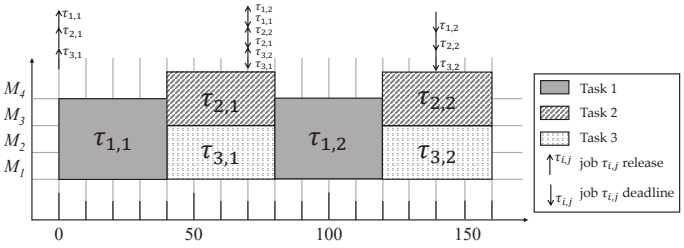


Fig. 1: Example gang task schedule.

A widely-accepted approach to reducing inter-application interference between applications co-executed upon a shared, multiprocessor/multicore platform is *gang scheduling* [1], [10], [8], [5], [2]. Gang scheduled tasks comprise k parallel threads simultaneously co-executed (i.e., in unison) on k different processing resources. The requirement of parallel task executing in unison is in contrast to traditional, multi-threaded parallel scheduling, where a parallel task may spawn multiple threads, and each thread (once released and ready for execution) will be scheduled independently of other threads of the same task (until a barrier/synchronization point is reached – e.g., a "join" phase of a fork-join task [19]). The intra-task dependencies make the execution flow of the traditional parallel task very complected at runtime, which may cause rather pessimistic schedulability loss. The requirement of unison execution for gang-scheduled tasks is attractive for system designers as it permits a parallel task use of low-overhead synchronization protocols for communicating between its tasks and to commence multiple threads

simultaneously. Especially, for data-intensive applications, such as vision-based autonomous driving systems and virtual reality systems, data is highly parallel and the computations can be accelerated by gang scheduling – all of an application's threads of execution being grouped into a gang and concurrently scheduled on distinct processors.

However, gang scheduling is a double-edged sword. A released gang job can be scheduled only if the number of idle processors is at least the number of processors required by the corresponding gang task. This simple constraint may introduce additional utilization loss for the multicore platform. Consider an intuitive example, where three gang tasks (τ_1, τ_2 and τ_3) are scheduled on four processors (*i.e.*, M_1, M_2, M_3, M_4 in Fig. 1) under global earliest-deadline-first (GEDF). τ_1 has a parallelism of three processors while both τ_2 and τ_3 have a parallelism of two. As seen in Fig. 1, at time 0, although there is an idle processor, the scheduler can schedule neither τ_2 nor τ_3 onto it because both tasks have a parallelism of two processors. From this task schedule, it is evident to see that the response time for each gang task increases over time but the multicore platform cannot be fully utilized due to the gang tasks' parallelism.

Gang tasks' non-malleable parallelism (as illustrated in Fig. 1) introduces a significant challenge to the derivation soft real-time (SRT) schedulability analysis [1]. For ordinary (non-parallel, independent) sporadic task systems, Devi [4] proved that the computing capacity of multicore platforms can be fully utilized, which means even if the total utilization of the task system is equal to the number of processors on the multicore platform, the real-time tasks will still have bounded response times. Unfortunately, as we mentioned above, the multicore platform cannot be fully utilized by the gang tasks. Existing results cannot be used directly to analyze the schedulability of gang tasks. To resolve this parallelism-induced research challenge, this work aims at developing techniques that can upper bound parallelism-induced utilization loss for any sporadic gang task systems and derive a corresponding utilization-based schedulability test. Specifically, we study gang task system scheduled on a homogeneous multiprocessor under the classical GEDF scheduling policy. To the best of our knowledge, this is the first utilization-based schedulability test derived for SRT sporadic gang task systems.

Related work. The problem of scheduling real-time gang tasks has received much recent attention in real-time systems community, e.g., [10], [8], [5], [2]. However, these efforts were directed on HRT systems only. On the other hand, since the seminal work by Devi and Anderson [4] on SRT systems that are defined by bounded tardiness, a series of work in this direction has been done, e.g., [3], [14], [6], which, however, address conventional sporadic (non-parallel) tasks only. To the best of our knowledge, the problem of gang scheduling for SRT systems has been tackled only in [20] where non-preemptive FIFO scheduling is considered. In contrast, we focus on preemptive GEDF scheduling in this paper.

Paper Contributions. This paper makes the following contributions to the soft real-time scheduling of gang-scheduled parallel sporadic tasks:

- We provide a motivating example (Section 3) to show that there exist sporadic gang task systems with utilization approaching one that cannot be scheduled by any algorithm (online or offline) on an M-processor platform to meet all deadlines. This motivates us to focus upon SRT gang scheduling of sporadic tasks.
- We introduce the concept of *parallelism-induced idleness* (Section 5) that refers to time intervals during which there is a ready gang task and available processors, but the task cannot execute due to the requirement that all threads of a task must execute in unison. Furthermore, we design an algorithm for quantifying the magnitude (i.e., number of processors) of the parallelism-induced idleness.
- We derive and prove the first *tardiness bounds* for gang-scheduled sporadic task systems under preemptive GEDF scheduling (Section 6). Under a total utilization bound, which serves as a SRT-schedulability test, these bounds are derived using a novel lag-based reasoning approach that incorporates the parallelism-induced idleness as a factor.
- We show that the required total utilization bound for ensuring such tardiness bounds is tight with respect to the platform capacity and maximum parallelism-induced idleness (Section 7).
- We empirically evaluate the effects of different degrees of task parallelism upon the SRT-schedulability of a sporadic gang task system (Section 8).

2 System Model

In this paper, we consider the sporadic gang task model, which extends the conventional sporadic task model by that each task may require and occupy *multiple*, instead of one, processors to commence any execution. Namely, we consider the problem of scheduling a set $\tau = \{\tau_1, ..., \tau_n\}$ of n independent sporadic gang tasks on M identical processors. Each task τ_i needs to occupy m_i available processors simultaneously for being scheduled to execute. m_i is called the *degree of parallelism* of τ_i. Similar to the conventional sporadic task model, each sporadic gang task τ_i releases a potentially infinite sequence of *jobs*, where arrival times of any two consecutive jobs of task τ_i must be separated by at least p_i time units. We also assume implicit deadlines in this paper, i.e., every job of task τ_i has an (absolute) deadline p_i time units after its release. We denote the j^{th} job of τ_i by $\tau_{i,j}$ and denote its release time and (absolute) deadline by $r_{i,j}$ and $d_{i,j}$, respectively, i.e.,

$$d_{i,j} = r_{i,j} + p_i. \tag{1}$$

Letting $f_{i,j}$ denote its *finish time*, the tardiness of job $\tau_{i,j}$ is defined by $\max\{f_{i,j} - d_{i,j}, 0\}$, and the tardiness of a task is defined by the maximum[2] tardiness over all its jobs. We also denote the worst-case execution time (WCET) of τ_i by e_i, i.e., each job of τ_i can execute at most e_i time units while occupying m_i available processors. Therefore, the worst-case *execution requirement* of task τ_i can be represented as

1. In SRT systems, deadlines of real-time tasks may be missed as long as the tardiness of every task is bounded (by task system parameters)

2. Or more precisely, it is "supremum" instead of "maximum" to better fit the fact that each task may release an *infinite* number of jobs.

Algorithm 1: Selecting Jobs to Schedule under GEDF

input : Ready(t), which is the ready job set at time t
output: Sched(t), which is the scheduled job set at time t

```
1 Sched(t) ← ∅
2 for each τ_{i,j} ∈ Ready(t) in deadline increasing order do
3     if Σ_{τ_{k,ℓ} ∈ Sched(t)} m_k ≤ M − m_i then
4         Sched(t) ← Sched(t) ∪ {τ_{i,j}}
5     end
6 end
```

an $m_i \times e_i$ rectangle in the schedule. Thus, each implicit-deadline sporadic gang task is specified by three parameters: $\tau_i = (e_i, m_i, p_i)$. A simple example of a real-time gang task system is given in Fig. 2. The parameters for each gang task are described in Example 1.

For each job $\tau_{i,j}$, we also call the preceding jobs (released prior to $\tau_{i,j}$) of the same task τ_i as the *predecessors* of $\tau_{i,j}$. At an arbitrary time instant t, a job is called *pending* if it is released but not completed at time t, and is called *ready* if it is pending and all its predecessors have been completed at time t. We also assume that only ready jobs can commence execution, i.e., at most one job of each task can be executed at a time.

Furthermore, the *utilization*[3] of each sporadic gang task τ_i is defined as

$$u_i = (e_i \cdot m_i)/p_i. \tag{2}$$

Note that the utilization of a gang task may be larger than one, which differs from the traditional sporadic task model. The utilization of the task system is $U_{sum} = \sum_{i=1}^{n} u_i$. While utilization is with respect to the *execution requirement* per task period, we also define

$$\lambda_i = e_i/p_i \tag{3}$$

with respect to the *execution time* per task period and λ_i is called the *horizontal utilization* [8] of task τ_i. By the definition of u_i and λ_i, it is clear that

$$\lambda_i = u_i/m_i. \tag{4}$$

Furthermore, because it is assumed that a job can commence execution after its release only when all its predecessors have completed, the jobs of each task must be executed in sequence while each individual job is executed in parallel on multiple processors. Therefore, it is clear that $\lambda_i \leq 1.0$ is a necessary condition for any task τ_i to possibly have bounded tardiness. This by Eq. (4) implies that $\forall i$, $u_i \leq m_i$ is necessary as well. Please note the conventional sporadic task model is a special case of the sporadic gang task model in this paper, where it happens that $\forall i, m_i = 1$ which implies $\forall i, u_i = \lambda_i$ as well.

In this paper, we focus on soft-real-time (SRT) systems where deadlines may be missed as long as the tardiness of every task is bounded (by task system parameters), i.e., no task has its tardiness grow job by job without bound potentially towards infinity. A task system is called *SRT-schedulable* under a particular scheduling algorithm if and only if every task is guaranteed bounded tardiness under this algorithm in all possible scenarios allowed by the task model. A task system is called *SRT-feasible* if and only if it is SRT-schedulable under some (potentially optimal) scheduling algorithm.

3. It is also called *rectangle utilization* in [8] in contrast to *horizontal utilization*. In this paper, references to "utilization" qualification should henceforth be taken to mean the rectangle utilization, whereas the horizontal one should always be explicitly specified for references.

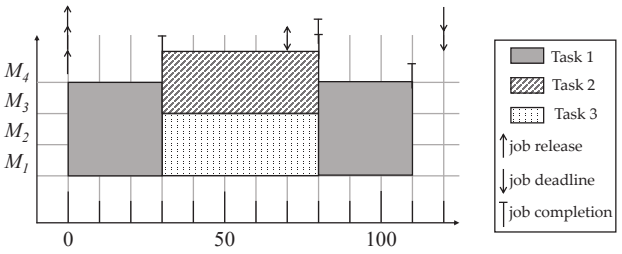


Fig. 2: An example gang task system.

We also focus on the preemptive GEDF scheduling in this paper. The priority of each job is determined by its deadline — the earlier the deadline, the higher the priority. We also assume deadline ties are broken arbitrarily but consistently, and therefore there is no *priority* ties while *deadline* ties may exist. Letting Ready(t) denote the set of *ready* jobs at an arbitrary time instant t, the set of jobs Sched(t) being scheduled at time t is determined by Algorithm 1. m_k is the degree of parallelism of τ_k, which has a job in Sched(t). Please note that, in practice, Algorithm 1 does not need to be evaluated at every time instant but only needs to be invoked when a job is completed and when a new job is released.

Example 1. *Figure 2 shows an example of scheduling a gang task system τ under GEDF on a four-processor system consisting of three gang tasks, $\tau_1 = (30,\ 3,\ 70)$, $\tau_2 = (50,\ 2,\ 120)$, and $\tau_3 = (50,\ 2,\ 120)$. All tasks release the first jobs at time instant 0 and according to GEDF, $\tau_{1,1}$ has the highest priority and starts executing when it is released. Since the degree of parallelism of both τ_2 and τ_3 is two, even though one processor is idle at time instant 0, $\tau_{2,1}$ and $\tau_{3,1}$ are delayed and start executing at time instant 30. Another interesting observation is that since $\tau_{1,1}$ has completed at time instant 30, by definition, $\tau_{1,2}$ is pending and ready at time instant 70 but preempted by $\tau_{2,1}$ and $\tau_{3,1}$ during the time interval $[70, 80)$. Therefore, $\tau_{1,2}$ starts executing at time instant 80.*

3 A Motivation Observation

Before presenting our developed analysis techniques in detail, we first intuitively motivate and explain the research challenges due to the parallelisms of gang tasks.

Example 2. *Consider a four-processor sporadic gang task set τ that consists of two gang tasks scheduled under preemptive GEDF: $\tau_1 = (\varepsilon,\ 4,\ 50)$, where $\varepsilon > 0$, and $\tau_2 = (50,\ 1,\ 50)$. Suppose both τ_1 and τ_2 release their first jobs at time instant 0, which is shown in Fig. 2. It is evident that on this preemptive GEDF schedule, the response times of both τ_1 and τ_2 increase over time. Because the degree of parallelism of τ_1 is four, which is equal to the number of processors, resulting in that jobs from τ_1 and τ_2 cannot execute concurrently. Because both tasks have a period of 50 time units and there is a combined WCET of $\varepsilon + 50$ every*

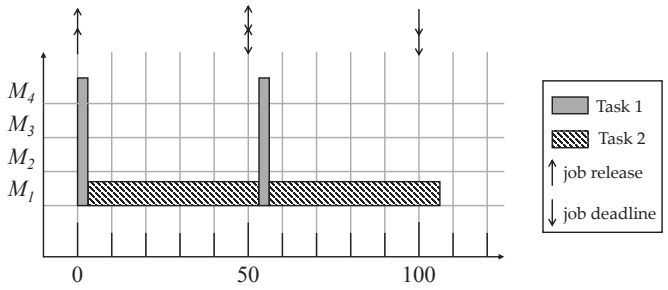


Fig. 3: An unschedulable gang task system.

period, the tardiness of at least one of the two tasks must increase without bound in the worst case, no matter what scheduling algorithm is applied. That is, this system is SRT-infeasible. An interesting observation from this example is that while τ_2 are executing, three processors are idle even τ_1 is pending, which means the computing capacity of the multiprocessor platform cannot be fully utilized. On the other hand, the total utilization (U_{sum}) of this task set is $\frac{4\varepsilon}{50} + \frac{50\times 1}{50} = 0.08\varepsilon + 1.0$, which is greater than but arbitrarily close to 1.0 when $\varepsilon \to 0^+$.

The above example shows that a *constant* utilization bound related to M only would not be very interesting — it cannot exceed 1.0, even if for any potentially optimal scheduler. Therefore, we will take further information from the task system parameters to obtain a more meaningful total utilization constraint for SRT-schedulability. in particular the Δ_i parameter for each task τ_i will be introduced and calculated later in Sec. 5.

Also, from above example, it is evident that the computing capacity of the multiprocessor platform cannot be fully utilized due to the gang tasks' parallelisms: for gang task scheduling, a released job can be scheduled only if the number of idle processors is at least the number of processors required by the corresponding gang task. Thus, at some time instants, a released gang job cannot be executed even some processors are idle. In other words, the computing capacity of the multiprocessor platform is partially lost, namely the utilization loss, due to the idleness. Such a worst case scenario is actually seen in the example shown in Fig. 3, where the second job of τ_1 is blocked by the first job of τ_2 at time instant 50 even though there are three idle processors.

A key insight to motivate our schedulability analysis. For any given gang task system, if we take the tasks' total utilization as the only parameter to validate its schedulability, the total utilization of the gang task system cannot be larger than 1, otherwise, the task system is unschedulable. Because each gang task may require multiple processors for execution at the same time and in the worst case, this requirement analytically converts the "multiprocessor scheduling" problem into a "single processor scheduling" problem (e.g., Example 2), enforcing the poor utilization. On the other hand, applications are defined using parallel gang structures to better exploit the parallel computing capability provided by the multicore platform. Example 3 shows a simple example to illustrate that a four processor platform is fully utilized by two gang tasks. Therefore, the proposed schedulability analysis should be able to take into account the specific degrees of the parallelisms for the gang tasks and allow gang task systems with bounded response time, as many as possible.

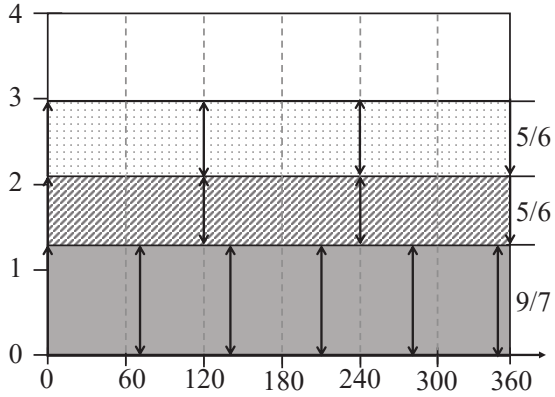


Fig. 4: An example PS schedule.

Example 3. *Consider a four-processor sporadic gang task set τ that consists of two gang tasks scheduled under preemptive GEDF: $\tau_1 = (25,\ 4,\ 50)$ and $\tau_2 = (25,\ 4,\ 50)$. Both tasks have zero tardiness and the total utilization of the task system is four.*

In light of the above discussion, intuitively, given a gang task system scheduled under GEDF, if the total utilization loss of the computing platform can be upper bounded by our analysis according to the tasks' specific parallelisms, then we can derive a practical utilization-based schedulability test for the gang task system. Since our analysis is based on the classic LAG-based reasoning, in the next section, we introduce some preliminaries first.

4 Preliminaries

Our approach towards determining tardiness bounds under preemptive EDF involves comparing the allocations to a concrete task system in a processor sharing (PS) schedule for τ and an actual preemptive GEDF schedule of interest for τ and quantifying the difference between the two. In this section, we provide necessary preliminaries for this approach.

4.1 Lag-based Reasoning

Definition 1. *For any given gang task system τ, a PS schedule is an ideal schedule where each task τ_i executes with a constant rate equal to u_i whenever it has a ready job. Furthermore, it is clear that in the PS schedule, every job is guaranteed to complete by its deadline and must complete exactly at its deadline if it executes for its worst-case execution requirement. Note that for a sporadic gang task system, u_i can be larger than 1.0. Fig. 4 shows an example PS schedule.*

Example 4. *The task system given in Example 1 contains three tasks, τ_1 with utilization $\frac{9}{7}$, τ_2 with utilization $\frac{5}{6}$, and τ_3 with utilization $\frac{5}{6}$. τ_1, τ_2, and τ_3 have a period of 70 time units, 120 time units, and 120 time units, respectively. Fig. 4 shows the PS schedule for this system where each task executes at a rate equal to its utilization.*

Our schedulability test is obtained by comparing the allocations to τ in the GEDF schedule $\mathcal{S}$ and the corresponding $\mathcal{PS}$ schedule, both on M processors, and quantifying the difference between the two. We analyze task allocations on a per-task basis.

Our analysis is based on the processing capacity allocations for the jobs, tasks, and the task set in the PS schedule and in the actual GEDF schedule. For any concrete instance[4] of the task set τ, we consider such allocations as follows. We let $\mathsf{A}(\tau_{i,j}, t_1, t_2, \mathcal{S})$ denote the total allocation to job $\tau_{i,j}$ in $\mathcal{S}$ in $[t_1, t_2)$. Then, the total time allocated to all jobs of τ_i in $[t_1, t_2)$ in $\mathcal{S}$ is given by

$$\mathsf{A}(\tau_i, t_1, t_2, \mathcal{S}) = \sum_{j \geq 1} \mathsf{A}(\tau_{i,j}, t_1, t_2, \mathcal{S}). \tag{5}$$

Letting $\mathcal{PS}$ denote the PS schedule, the difference between the allocation to a job $\tau_{i,j}$ in $\mathcal{PS}$ and $\mathcal{S}$ during time interval $[0, t)$ is called the lag of job $\tau_{i,j}$ at time t in schedule $\mathcal{S}$ and is defined by

$$\mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) = \mathsf{A}(\tau_{i,j}, 0, t, \mathcal{PS}) - \mathsf{A}(\tau_{i,j}, 0, t, \mathcal{S}). \tag{6}$$

Example 5. *Figure 2 shows an example of scheduling a gang task system τ under GEDF on a four-processor system consisting of three gang tasks. The corresponding PS schedule is given in Figure 3. According to Eq. 6,*

$$\begin{aligned} \mathsf{lag}(\tau_{2,1}, 50, \mathcal{S}) &= \mathsf{A}(\tau_{2,1}, 0, 50, \mathcal{PS}) - \mathsf{A}(\tau_{2,1}, 0, 50, \mathcal{S}) \\ &= \frac{5}{6} \times 50 - 2 \times 20 = \frac{5}{3}. \end{aligned} \tag{7}$$

Also, the lag of task τ_i at time t in schedule $\mathcal{S}$ is defined by

$$\begin{aligned} \mathsf{lag}(\tau_i, t, \mathcal{S}) &= \sum_{j \geq 1} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) \\ &= \sum_{j \geq 1} (\mathsf{A}(\tau_{i,j}, 0, t, \mathcal{PS}) - \mathsf{A}(\tau_{i,j}, 0, t, \mathcal{S})). \end{aligned} \tag{8}$$

Furthermore, the LAG of the entire task set τ at time t in schedule $\mathcal{S}$ is defined by

$$\mathsf{LAG}(\tau, t, \mathcal{S}) = \sum_{\tau_i \in \tau} \mathsf{lag}(\tau_i, t, \mathcal{S}). \tag{9}$$

5 Idleness-induced Utilization Loss

In this section, before taking a further step to upper bound the utilization loss on the GEDF schedule, we introduce two types of idleness first using the example in Fig. 5.

Definition 2. *Normally, there are two types of idleness on the GEDF schedule of gang tasks, **parallelism-induced idleness** and **natural idleness**. A time instant t is parallelism-induced idle for a job set J if (i) at least one processor is idle at t and (ii) at least one gang job form J is pending but does not execute at t. A time instant t is naturally idle for a job set J if (i) at least one processor is idle at t and (ii) all pending gang jobs form J are executing at t. A time interval is parallelism-induced idle (resp. naturally idle) for J if each instant within it is parallelism-induced idle (resp. naturally idle) for J.*

4. It means a set of concrete job release times and actual execution times that follow the task model specification of every task.

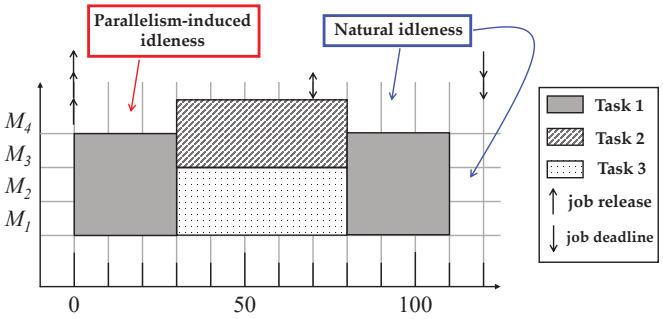


Fig. 5: Parallelism-induced idleness VS. Natural idleness.

Example 6. *Both types of idleness exist in the schedule of Example 1, which is shown in Fig. 5. The idleness from 0 to 30 is parallelism-induced idleness, since both $\tau_{2,1}$ and $\tau_{3,1}$ are released at 0 but do execute during $[0,\ 30)$. The idleness from 80 to 120 is natural idleness, because all pending jobs execute during $[80,\ 120)$.*

Intuitively, for soft real-time GEDF scheduling, the natural idleness does not hurt the schedulability of real-time gang tasks, which can be illustrated by the following claim.

Claim 1. *For any given real-time gang task system scheduled on M identical processors under GEDF, if there is no parallelism-induced idleness on the schedule, the task system is schedulable (i.e., each task has bounded tardiness) even if its total utilization is M (and at most M).*

The correctness of this claim will be formally discussed in Sec. 7. Essentially, upper bounding the parallelism-induced idleness on the schedule can yield a utilization-based schedulability test for gang tasks. In order to achieve this goal, in the rest of this section, we first calculate the upper bound on the number of idle processors during any parallelism-induced idle interval for an arbitrary gang task τ_i.

Definition 3. *Let I_t denote the number of idle processors at time instant t. Thus, at time instant t, $M - I_t$ processors are busy executing jobs.*

Definition 4. *Let Δ_i denote the maximum possible number of idle processors at any time during τ_i's non-executing intervals in which τ_i has pending jobs but does not execute. In other words, if a gang job $\tau_{i,j}$ is released before but does not execute during a non-executing interval of τ_i, the number of idle processing units during this non-executing interval is at most Δ_i.*

Finding Δ_i. According to Definition 4, setting Δ_i to be $m_i - 1$ is safe but can be too pessimistic, which will result in a less efficient schedulability test. We now present a polynomial time algorithm based on dynamic programming, which finds Δ_i through exploring the specific tasks' parallelism characteristics.

In order to calculate Δ_i, we need to find a subset of tasks in $\tau \setminus \tau_i$ satisfying the following two properties: (*i*) the total degree of parallelism of tasks in this subset is at least $M - m_i + 1$, and (*ii*) the total degree of parallelism of tasks in this subset is the smallest one among all subsets satisfying property (*i*). The first property guarantees that the total degree of parallelism of all tasks in the task set is

Algorithm 2: Δ_i identification algorithm.

```
input : M, m_1, m_2, ..., m_n
output: Δ_i
1  N = m_i − 1, Δ_i = n.
2  if Σ_{i=1}^{n} m_i ≤ M then
3  |  τ is schedulable, and Δ_i does not exist.
4  end
5  else
6  |  for q = 1 → n − 1 do
7  |  |  if q < i then
8  |  |  |  z_q = m_q
9  |  |  end
10 |  |  if q ≥ i then
11 |  |  |  z_q = m_{q+1}
12 |  |  end
13 |  end
14 |  for q = 1 → n − 1 do
15 |  |  Δ̄[0][q] = 0.
16 |  end
17 |  while N ≠ Δ_i do
18 |  |  for a = 1 → n − 1 do
19 |  |  |  for b = 1 → M − N do
20 |  |  |  |  Δ̄[a][b] = 0
21 |  |  |  |  if z_a ≤ b then
22 |  |  |  |  |  if Δ̄[a − 1][b − z_a] + z_a ≥ Δ̄[a − 1][b]
   |  |  |  |  |   then
23 |  |  |  |  |  |  Δ̄[a][b] = Δ̄[a − 1][b − z_a] + z_a
24 |  |  |  |  |  end
25 |  |  |  |  |  else
26 |  |  |  |  |  |  Δ̄[a][b] = Δ̄[a − 1][b]}
27 |  |  |  |  |  end
28 |  |  |  |  end
29 |  |  |  end
30 |  |  end
31 |  |  Δ_i = M − Δ̄[n − 1][M − N].
32 |  |  if N ≠ Δ_i then
33 |  |  |  N = N − 1
34 |  |  end
35 |  end
36 end
```

large enough to preempt τ_i on M processors; the second property ensures us to identify the minimum total degree of parallelism (thus the maximum possible number of idle processors under all scenarios) during τ_i's non-executing intervals.

Algorithm description. We develop a polynomial-time algorithm that applies a dynamic programming approach to reduce the complexity of finding the subset of tasks that exhibits the smallest degree of parallelism from all possible subsets. The detailed pseudocode of this algorithm is described by Algorithm 2, which is given in the appendix of [5] as well. The basic idea behind this algorithm can be explained as follows. First we use dynamic programming to check whether there exists a subset of tasks in $\tau \backslash \tau_i$ satisfying that the total degree of parallelism of tasks in this subset is $M-m_i+1$. If yes, $\Delta_i = m_i-1$; otherwise, we check whether there exists a subset of tasks in $\tau \setminus \tau_i$ satisfying that the total degree of parallelism of tasks in this subset is $M - m_i + 2$. We continue this iteration process until we find Δ_i. Note that the dynamic programming is nearly identical to the knapsack problem: each task τ_i corresponds to an item with both size and value equal to m_i; the size of the knapsack is initially set to $M - m_i + 1$. A run of the standard dynamic programming formulation for knapsack will require $\mathcal{O}(Mn)$ time to check if there is a subset with value exactly equal to $M-m_i+1$. Since we have to run this for difference knapsack sizes, the total time to find Δ_i would also be $\mathcal{O}(M^2n)$. Since the total degree of parallelism of all tasks in τ is at least M, Δ_i can always be found.

Example 7. *We perform this algorithm on a specific gang task system for example. Consider a ten-processor sporadic gang task set τ that consists of 5 gang tasks scheduled under preemptive GEDF: $m_1 = 4$, $m_2 = 4$, $m_3 = 4$, $m_4 = 5$, $m_5 = 5$. Δ_5 denotes the maximum possible number of idle processors at any time during τ_i's non-executing intervals in which τ_i have pending jobs but does not execute. In step 1, $\Delta_5 = m_5 - 1 = 4$ and we use dynamic programming to check whether there exists a subset of tasks in $\tau \backslash \tau_5$ satisfying that the total degree of parallelism of tasks in this subset is $M - m_5 + 1 = 6$. The answer is no, according to Algorithm 2 (from line 17 to line 35), then $\Delta_5 = 3$. In step 2, we check whether there exists a subset of tasks in $\tau \setminus \tau_5$ satisfying that the total degree of parallelism of tasks in this subset is 7. It is evident that the answer is no. Then, according to Algorithm 2, $\Delta_5 = 2$ and we can find that the total degree of parallelisms of τ_1 and τ_2 is 8. Therefore, in light of Algorithm 2 (line 17), we have $\Delta_5 = 2$.*

6 Tardiness Bounds

In this section, we derive a tardiness bound for each SRT task τ_i, given that the following total utilization constraint holds:

$$U_{sum} \leq M - \Delta_{\max}, \tag{10}$$

where

$$\Delta_{\max} = \max_{\tau_i \in \tau}\{\Delta_i\}.$$

Overview. Before diving into the technical details, we perform a proof overview first to discuss the intuition behind the proof. We will prove that each task τ_i has a tardiness bound of $x + e_i$ under GEDF scheduling, where

$$x = \max\left\{\frac{(M - \Delta_{\max} - 1)e_{\max} - e_{\min}}{(M - \Delta_{\max})(1 - \lambda_{\max}) + \lambda_{\max}}, 0\right\}, \tag{11}$$

$$\lambda_{\max} = \max_{\tau_i \in \tau}\{\lambda_i\}, e_{\max} = \max_{\tau_i \in \tau}\{e_i\}, \text{ and } e_{\min} = \min_{\tau_i \in \tau}\{e_i\}.$$

This proof is by contradiction. We suppose a tardiness bound does not hold, and let $\tau_{k,\ell}$ denote the job with the earliest deadline that breaks its tardiness bounds. We also denote the absolute deadline of $\tau_{k,\ell}$ by t_d (*i.e.*, $d_{k,\ell} = t_d$). That is, $\tau_{k,\ell}$ has not completed by $t_d + x + e_k$, while any other job $\tau_{i,j}$ with higher priority than $\tau_{k,\ell}$ under GEDF have tardiness at most $x + e_i$. Please note that we assume deadline ties are broken arbitrarily but consistently under GEDF scheduling. In the rest of this section, we will derive a contradiction with the definition of x in Eq. (11). The contradiction implies that the supposition cannot be true, and therefore a tardiness bound of $x + e_i$ for every task τ_i must hold.

Because of the *preemptive* GEDF scheduling rule (G) in Sec. 2, only jobs with absolute deadlines at or before time t_d

may have an impact, directly or indirectly, on the execution of job $\tau_{k,\ell}$. Therefore, letting Υ denote the set of jobs with absolute deadlines at or before time t_d, it suffices to consider jobs in Υ only in the rest of this section. That is, without loss of generality, all jobs with deadline later than t_d are considered as removed from the GEDF schedule we are investigating as their removal will not change the deadline miss that arises at t_d.

Then, we let $W(t)$ denote the total *pending* workload (by jobs in Υ) at time t in the GEDF schedule $\mathcal{S}$ and a contradiction will be obtained in three steps: (*i*) derive an upper bound on $W(t_d)$ by the total utilization constraint; (*ii*) derive a lower bound on $W(t_d)$ by the supposition that job $\tau_{k,\ell}$ has tardiness greater than $x + e_k$; and (*iii*) the first two steps will imply a condition that x must satisfy and this condition will directly contradict the definition of x in Eq. (11).

For ease of reading, we summarize, by Table. 1, the notations defined in the previous sections as well as those to be introduced in this section.

Symbol	Meaning
τ	Task set
n	Number of tasks in τ
M	Number of processors
τ_i	i^{th} task in τ
m_i	Degree of parallelism of τ_i
p_i	Period of τ_i
e_i	Worst-case execution time (WCET) of τ_i
u_i	(Rectangle) utilization of τ_i
λ_i	Horizontal utilization of τ_i
$\tau_{i,j}$	j^{th} job of τ_i
$r_{i,j}$	Release time of $\tau_{i,j}$
$d_{i,j}$	Absolute deadline of $\tau_{i,j}$
$f_{i,j}$	Finish time of $\tau_{i,j}$
t	An arbitrary time instant
A	Processing capability allocation
lag	"Lag" of a task
LAG	"Lag" of the entire system
$\mathcal{S}$	GEDF schedule
$\mathcal{PS}$	Processor sharing (PS) schedule
Δ_i	Maximum possible number of idle processors at any time when τ_i has pending jobs but does not execute.
$\tau_{k,\ell}$	Job of interest being analyzed
t_d	Absolute deadline of $\tau_{k,\ell}$
x	Part of the tardiness bound expression and defined by Eq. (11)
y	Length of gang-busy time interval after t_d, to be used and more formally defined in the proof of Lemma 7
Υ	Set of jobs with absolute deadlines at or before time t_d
$W(t)$	Total pending workload by jobs in Υ at time t in the GEDF schedule $\mathcal{S}$
Z	Temporal notation in the proofs for certain accumulated actual execution requirement

TABLE 1: Notation Summary.

Definition 5. *A time instant t is called* gang-busy *if at least $(M - \Delta_{\max})$ processing units execute jobs in Υ at time t and is called* gang-idle *if at most $(M - \Delta_{\max} - 1)$ processing units execute jobs in Υ (i.e., at least $(\Delta_{\max} + 1)$ processing units are "idle" at time t. A time interval is gang-busy (gang-idle, respectively) for Υ if every instant within the time interval is gang-busy (gang-idle, respectively) for Υ.*

6.1 An upper bound on $W(t_d)$

We first derive upper bound on the lag of each individual task at a certain time instant t by either of the following two lemmas, depending on whether the task has pending jobs at time t.

Lemma 1. *For each task τ_i and for all $t \leq t_d$, if task τ_i does have pending jobs at time t, then*

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq m_i(\lambda_i \cdot x + e_i). \tag{12}$$

Proof. As task τ_i does have pending jobs at time t, we let $\tau_{i,j}$ denote the ready one, i.e., all predecessors of $\tau_{i,j}$ have completed by time t. Thus, letting Z denote the accumulated actual *execution requirement* by the jobs of τ_i prior to job $\tau_{i,j}$ and letting $\delta \in [0, e_i)$ denote the number of *time units* that job $\tau_{i,j}$ has been executed by time t, we have that

$$\mathsf{A}(\tau_i, 0, t, \mathcal{S}) = Z + \delta \cdot m_i.$$

On the other hand, in the PS schedule, all jobs of τ_i prior to job $\tau_{i,j}$ must have completed by time $r_{i,j}$ and then $\tau_{i,j}$ is executed at the rate of u_i, starting from $r_{i,j}$. Therefore,

$$\mathsf{A}(\tau_i, 0, t, \mathcal{PS}) = Z + (t - r_{i,j}) \cdot u_i.$$

As a result, we have

$$\begin{aligned} \mathsf{lag}(\tau_i, t, \mathcal{S}) &= \mathsf{A}(\tau_i, 0, t, \mathcal{PS}) - \mathsf{A}(\tau_i, 0, t, \mathcal{S}) \\ &= (t - r_{i,j}) \cdot u_i - \delta \cdot m_i \end{aligned} \tag{13}$$

We then discuss two cases by $d_{i,j}$, which is the absolute deadline of job $\tau_{i,j}$.

Case 1: $d_{i,j} \geq t$**.** In this case, by Eq. (13)

$$\begin{aligned} \mathsf{lag}(\tau_i, t, \mathcal{S}) &= (t - r_{i,j}) \cdot u_i - \delta \cdot m_i \\ &\leq \{\text{because } \delta \geq 0 \text{ and } m_i \geq 1\} \\ &\quad (t - r_{i,j}) \cdot u_i \\ &\leq \{\text{because in Case 1, } d_{i,j} \geq t\} \\ &\quad (d_{i,j} - r_{i,j}) \cdot u_i \\ &= \{\text{by Eq. (1)}\} \\ &\quad p_i \cdot u_i \\ &= \{\text{by Eq. (2)}\} \\ &\quad m_i \cdot e_i \end{aligned} \tag{14}$$

Case 2: $d_{i,j} < t$**.** Because $\tau_{k,\ell}$ is the job with the earliest deadline that breaks the tardiness bounds and $d_{i,j} < t \leq t_d$ in this case, the job $\tau_{i,j}$ must be able to complete by $d_{i,j} + x + e_i$. On the other hand, recall that $\delta \in [0, e_i)$ denote the number of time units that $\tau_{i,j}$ has completed by time t, $\tau_{i,j}$ is only able to complete at or after $t + e_i - \delta$, when $\tau_{i,j}$ executes for its worst-case execution time. Therefore, we have

$$t + e_i - \delta \leq d_{i,j} + x + e_i,$$

which implies

$$t \le d_{i,j} + x + \delta. \tag{15}$$

Thus, by Eq. (13)

$$\begin{aligned}
\mathsf{lag}(\tau_i, t, \mathcal{S}) &= (t - r_{i,j}) \cdot u_i - \delta \cdot m_i \\
&\le \{\text{by Eq. (15)}\} \\
&\quad (d_{i,j} + x + \delta - r_{i,j}) \cdot u_i - \delta \cdot m_i \\
&= \{\text{by Eq. (1)}\} \\
&\quad (p_i + x + \delta) \cdot u_i - \delta \cdot m_i \\
&= \{\text{rearrange}\} \\
&\quad (p_i + x) \cdot u_i + \delta \cdot (u_i - m_i) \\
&\le \{\text{because } \delta \ge 0 \text{ and } u_i \le m_i\} \\
&\quad (p_i + x) \cdot u_i \\
&= \{\text{by Eq. (2) and Eq. (4)}\} \\
&\quad m_i \cdot e_i + x \cdot \lambda_i \cdot m_i \\
&= \{\text{rearrange}\} \\
&\quad m_i(\lambda_i \cdot x + e_i)
\end{aligned} \tag{16}$$

Thus, combining Cases 1 and 2 and by Eq. (14) and Eq. (16), the lemma follows.

□

Lemma 2. *If task τ_i has no pending job at time t,*

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) \le 0. \tag{17}$$

Proof. Let Z denote the accumulated actual *execution requirement* by the jobs of task τ_i that have been released at or before time t. Because task τ_i has no pending job at time t, we have

$$\mathsf{A}(\tau_i, 0, t, \mathcal{S}) = Z \tag{18}$$

trivially. On the other hand, by definition, only jobs that have been released may be executed in the PS schedule, therefore we have

$$\mathsf{A}(\tau_i, 0, t, \mathcal{PS}) \le Z. \tag{19}$$

Thus, combining Eq. (18) and Eq. (19) together, according to Eq. (6), $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\tau_i, 0, t, \mathcal{PS}) - \mathsf{A}(\tau_i, 0, t, \mathcal{S}) \le 0$. The lemma follows. □

We then derive an upper bound on $\mathsf{LAG}(\tau, t, \mathcal{S})$ such that $t < t_d$ and t is gang-idle, by the supposition that $\tau_{k,\ell}$ denote the job with highest priority that breaks the tardiness bounds.

Lemma 3. *For all $t \le t_d$, if time instant t is gang-idle, then*

$$\mathsf{LAG}(\tau, t, \mathcal{S}) \le (M - \Delta_{\max} - 1) \cdot (\lambda_{\max} \cdot x + e_{\max}). \tag{20}$$

Proof. Letting γ denote the set of tasks that have pending jobs at time t, time instant t being gang-idle implies that

$$\sum_{\tau_i \in \gamma} m_i \le M - \Delta_{\max} - 1. \tag{21}$$

Also, all tasks in γ must be scheduled for executing at time t, and all tasks in $\tau \setminus \gamma$ must have no pending job at time t. Otherwise, the fact that $\Delta_{\max} + 1$ processing units are idle at time t would contradict the definition of $\Delta_{\max}$. Thus,

$$\begin{aligned}
\mathsf{LAG}(\tau, t, \mathcal{S}) &= \sum_{\tau_i \in \tau} \mathsf{lag}(\tau_i, t, \mathcal{S}) \\
&= \{\text{rearrange}\} \\
&\quad \sum_{\tau_i \in \gamma} \mathsf{lag}(\tau_i, t, \mathcal{S}) + \sum_{\tau_i \in \tau \setminus \gamma} \mathsf{lag}(\tau_i, t, \mathcal{S}) \\
&\le \{\text{by Lemma 1 and Lemma 2, respectively}\} \\
&\quad \sum_{\tau_i \in \gamma} m_i(\lambda_i \cdot x + e_i) + \sum_{\tau_i \in \tau \setminus \gamma} 0 \\
&\le \{\text{because } \lambda_i \le \lambda_{\max} \text{ and } e_i \le e_{\max} \text{ for all } i\} \\
&\quad \sum_{\tau_i \in \gamma} m_i(\lambda_{\max} \cdot x + e_{\max}) \\
&\le \{\text{by Eq. (21)}\} \\
&\quad (M - \Delta_{\max} - 1) \cdot (\lambda_{\max} \cdot x + e_{\max}).
\end{aligned}$$

The lemma follows. □

Next, we prove the following lemma to show that the system-wide LAG cannot increase through a gang-busy time interval.

Lemma 4. *For any $t_1 < t_2$ such that time interval $[t_1, t_2)$ is gang-busy, if Eq. (10) holds, then $\mathsf{LAG}(\tau, t_1, \mathcal{S}) \ge \mathsf{LAG}(\tau, t_2, \mathcal{S})$.*

Proof. Because in the PS schedule each task τ_i is executed at a constant rate of u_i if it has pending job(s),

$$\mathsf{A}(\tau_i, t_1, t_2, \mathcal{PS}) \le u_i \cdot (t_2 - t_1).$$

Therefore,

$$\sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, t_1, t_2, \mathcal{PS}) \le \sum_{\tau_i \in \tau} u_i \cdot (t_2 - t_1) = U_{sum} \cdot (t_2 - t_1). \tag{22}$$

On the other hand, because time interval $[t_1, t_2)$ is gang-busy, we have

$$\sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, t_1, t_2, \mathcal{S}) \ge (M - \Delta_{\max}) \cdot (t_2 - t_1). \tag{23}$$

Thus,

$$\begin{aligned}
&\mathsf{LAG}(\tau, t_2, \mathcal{S}) \\
&= \mathsf{LAG}(\tau, t_1, \mathcal{S}) + \sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, t_1, t_2, \mathcal{PS}) - \sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, t_1, t_2, \mathcal{S}) \\
&\le \{\text{by Eq. (22) and Eq. (23)}\} \\
&\quad \mathsf{LAG}(\tau, t_1, \mathcal{S}) + U_{sum} \cdot (t_2 - t_1) - (M - \Delta_{\max}) \cdot (t_2 - t_1) \\
&= \{\text{rearrange}\} \\
&\quad \mathsf{LAG}(\tau, t_1, \mathcal{S}) - (M - \Delta_{\max} - U_{sum}) \cdot (t_2 - t_1) \\
&\le \{\text{by Eq. (10)}\} \\
&\quad \mathsf{LAG}(\tau, t_1, \mathcal{S}).
\end{aligned}$$

The lemma follows. □

We finally have an upper bound on $W(t_d)$ by proving the following lemma.

Lemma 5. *If Eq. (10) holds, then*

$$W(t_d) \le (M - \Delta_{\max} - 1) \cdot (\lambda_{\max} \cdot x + e_{\max}). \tag{24}$$

Proof. Recall that, in this section we only consider jobs with deadlines at or before t_d, i.e., jobs in Υ, and all other

jobs are viewed as removed. By the definition of the PS schedule, all jobs in Υ must complete by their deadlines and therefore must complete by t_d. As a result, the total amount of workload that could contribute to $W(t_d)$ is $\sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, 0, t_d, \mathcal{PS})$. On the other hand, such workload that has been completed in schedule $\mathcal{S}$ by time t_d, by definition, is $\sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, 0, t_d, \mathcal{S})$. Therefore,

$$\begin{aligned} W(t_d) &= \sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, 0, t_d, \mathcal{PS}) - \sum_{\tau_i \in \tau} \mathsf{A}(\tau_i, 0, t_d, \mathcal{S}) \\ &= \mathsf{LAG}(\tau, t_d, \mathcal{S}). \end{aligned} \tag{25}$$

We let t_0 denote the latest gang-idle time instant at or before t_d, or $t_0 = 0$ if no such time instant exists (i.e., $[0, t_d)$ is a gang-busy time interval). If $t_0 > 0$, by Lemma 3, we have

$$\mathsf{LAG}(\tau, t_0, \mathcal{S}) \leq (M - \Delta_{\max} - 1) \cdot (\lambda_{\max} \cdot x + e_{\max}), \tag{26}$$

and if $t_0 = 0$, Eq. (26) above is trivially true as $\mathsf{LAG}(\tau, 0, \mathcal{S}) = 0$.

Meanwhile, because $[t_0, t_d)$ is a gang-busy time interval in either case and Eq. (10) holds, by Lemma 4, we have

$$\mathsf{LAG}(\tau, t_0, \mathcal{S}) \geq \mathsf{LAG}(\tau, t_d, \mathcal{S}) \tag{27}$$

Thus, by Eq. (25), Eq. (26), and Eq. (27), the lemma follows. □

6.2 A lower bound on $W(t_d)$

On the other hand, we next will derive a lower bound on $W(t_d)$ such that job $\tau_{k.\ell}$ could still have incomplete work at time $d_{k,\ell} + x + e_k$.

Lemma 6. *Once the system becomes gang-idle at or after time t_d, it remains gang-idle and all tasks with pending job(s) must be scheduled to be executed and must be continuously executed until all of its jobs (in Υ) have completed.*

Proof. Because at least $\Delta_{\max} + 1$ processing units are idle at a gang-idle time instant, all tasks with pending jobs must be scheduled to be executed at such time instant. Also, it is impossible to release any new job in Υ at or after time t_d. Therefore, once the system becomes gang-idle at or after time t_d, it will remains gang-idle then. The lemma follows. □

Lemma 7. *If $\tau_{k,\ell}$ has not completed by $t_d + x + e_k$, it must hold that*

$$W(t_d) > (M - \Delta_{\max}) \cdot x + e_k. \tag{28}$$

Proof. We prove by contrapositive. That is, supposing $W(t_d) \leq (M - \Delta_{\max}) \cdot x + e_k$, we show that $\tau_{k,\ell}$ must complete by time $t_d + x + e_k$. By Lemma 6, from time t_d, the system must be gang-busy within time interval $[t_d, t_d + y)$ and then be gang-idle within time interval $[t_d + y, +\infty)$ for some $y \geq 0$.

If $m_k \geq M - \Delta_{\max}$, the system keeps gang-busy after t_d as long as $\tau_{k,\ell}$ has not completed, because either task τ_k being executing or the ready job of τ_k being prevented from executing due to the lack of available processing units implies the system is gang-busy. That is, $\tau_{k,\ell}$ must complete by $t_d + y$. Also, $W(t_d) \leq (M - \Delta_{\max}) \cdot x + e_k$ implies that $y \leq x + \frac{e_k}{M - \Delta_{\max}} \leq x + e_k$. Therefore, $\tau_{k,\ell}$ must complete by $t_d + x + e_k$.

In the rest of this proof, we focus on the other case of m_k that

$$m_k < M - \Delta_{\max}, \tag{29}$$

and we discuss the following two cases of such y.

Case 1: $y \leq x$**.** Because $[t_d + y, +\infty)$ is gang-idle and $t_d + x \in [t_d + y, +\infty)$ in this case, task τ_k must have been continuously executing since time $t_d + x$ if it is not completed by this time. In this case, only the predecessor jobs of the same task may prevent job $\tau_{k,\ell}$ from continuously execution from time $t_d + x$. However, the predecessor jobs of the same task may prevent job $\tau_{k,\ell}$ must have a deadline at or before $t_d - p_k$ and tardiness at most $x + e_k$. Therefore, any such predecessor job must have completed by $t_d - p_k + x + e_k \leq t_d + x$ due to $e_k \leq p_k$. Thus, job $\tau_{k,\ell}$ must have been continuously executing since time $t_d + x$ and consequently must complete by time $t_d + x + e_k$.

Case 2: $y > x$**.** Because $[t_d + y, +\infty)$ is gang-idle, task τ_k must have been continuously executing since time $t_d + y$. Letting z denote the number of time units for which task τ_k has been continuously executing since $t_d + y$, i.e. $\tau_{k,\ell}$ must complete by time $t_d + y + z$, the total workload completed during $[t_d, +\infty)$ is at least

$$(M - \Delta_{\max}) \cdot y + m_k \cdot z,$$

which cannot exceed the total pending workload at t_d, i.e., $W(t_d)$. On the other hand, the contrapositive of this lemma supposed that $W(t_d) \leq (M - \Delta_{\max}) \cdot x + e_k$. Therefore,

$$(M - \Delta_{\max}) \cdot y + m_k \cdot z \leq (M - \Delta_{\max}) \cdot x + e_k.$$

That is,

$$z \leq \frac{(M - \Delta_{\max}) \cdot (x - y) + e_k}{m_k} \tag{30}$$

Thus,

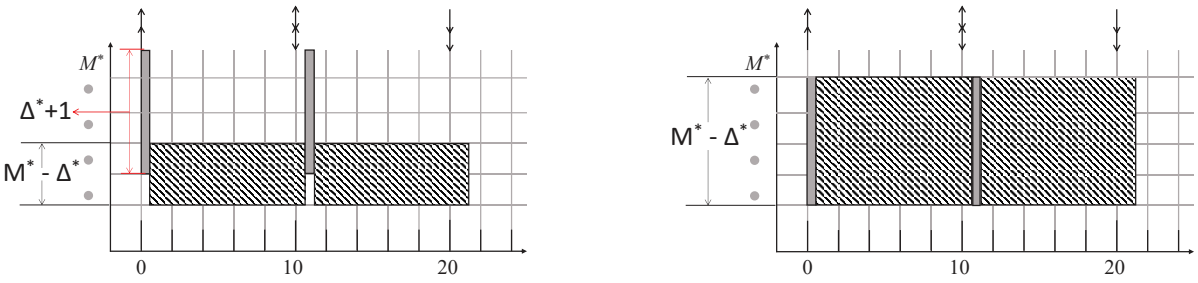
$$\begin{aligned} y + z &\leq y + \frac{(M - \Delta_{\max}) \cdot (x - y) + e_k}{m_k} \\ &= \frac{(M - \Delta_{\max}) \cdot (x - y) + m_k \cdot y + e_k}{m_k} \\ &= \frac{(M - \Delta_{\max} - m_k) \cdot (x - y) + m_k \cdot x + e_k}{m_k} \\ &= \frac{(M - \Delta_{\max} - m_k) \cdot (x - y)}{m_k} + x + \frac{e_k}{m_k} \\ &\leq \{\text{by Eq. (29) and } y > x \text{ in Case 2}\} \\ &\quad x + \frac{e_k}{m_k} \\ &\leq \{\text{because } m_k \geq 1\} \\ &\quad x + e_k. \end{aligned}$$

Since $\tau_{k,\ell}$ must complete by time $t_d + y + z$, it must complete by time $t_d + x + e_k$ as well.

Combining Case 1 and Case 2, the lemma follows. □

6.3 Finishing up

In the above two subsections, we have shown given the supposition that a tardiness bound of $x + e_i$ for each task τ_i does not hold, such a job $\tau_{k,\ell}$ that does not complete by $t_d + x + e_k$ must exist and then both Lemma 5 and Lemma 7 must hold. As shown in the following theorem, this in



(a) Case 1: $\Delta^* \geq \frac{M^*-1}{2}$. (b) Case 2: $\Delta^* < \frac{M^*-1}{2}$.

Fig. 6: For any M and $\Delta_{\max}$, there exists an SRT-infeasible system with total utilization greater than but arbitrarily close to $(M - \Delta_{\max})$.

fact leads to a contradiction, and therefore the supposition cannot be true. Thus, each task τ_i has a tardiness bound of $x + e_i$.

Theorem 1. *If Eq.* (10) *holds, any task τ_i cannot have tardiness greater than $x + e_i$, where x is defined by Eq.* (11)*.*

Proof. Suppose some tasks do have tardiness exceeding the claimed bound above. Recall that we let $\tau_{k,\ell}$ denote the job with the earliest deadline that has tardiness greater than $x + e_k$ and let t_d denote the absolute deadline of $\tau_{k,\ell}$. By Lemma 5, we have

$$W(t_d) \leq (M - \Delta_{\max} - 1) \cdot (\lambda_{\max} \cdot x + e_{\max}),$$

and by Lemma 7, we have

$$W(t_d) > (M - \Delta_{\max}) \cdot x + e_k.$$

Therefore, they imply that

$$(M - \Delta_{\max}) \cdot x + e_k < (M - \Delta_{\max} - 1) \cdot (\lambda_{\max} \cdot x + e_{\max}),$$

which implies that

$$x < \frac{(M - \Delta_{\max} - 1)e_{\max} - e_k}{(M - \Delta_{\max})(1 - \lambda_{\max}) + \lambda_{\max}}. \tag{31}$$

Because it is clear that $e_k \geq e_{\min}$ by definition, Eq. (31) implies that

$$x < \frac{(M - \Delta_{\max} - 1)e_{\max} - e_{\min}}{(M - \Delta_{\max})(1 - \lambda_{\max}) + \lambda_{\max}}. \tag{32}$$

On the other hand, x is defined by Eq. (11), which implies that

$$x \geq \frac{(M - \Delta_{\max} - 1)e_{\max} - e_{\min}}{(M - \Delta_{\max})(1 - \lambda_{\max}) + \lambda_{\max}}. \tag{33}$$

It is clear that Eq. (32) and Eq. (33) are a contradiction. Thus, the supposition at the beginning of this proof cannot be true, and the theorem follows. □

7 Discussions

As indicated by Eq. (10), a total utilization bound of $(M - \Delta_{\max})$ is required for guaranteeing the tardiness bounds provided in the prior section. This implies a potential utilization loss of $\Delta_{\max}$ in the system with respect to the total computing capacity provided by the M processing units.

In this section, we discuss such utilization loss from two aspects. First, we show that such utilization loss is because of parallelism-induced idleness. Second, we show that for any M and $\Delta_{\max}$, there exists a system with total utilization greater than but arbitrarily close to $M - \Delta_{\max}$ that is not SRT-schedulable under *any* algorithm, i.e., it is not SRT-feasible.

As claimed in Claim 1 in Sec. 5, existence of parallelism-induced idleness is the cause of the utilization loss. On the other hand, non-existence of parallelism-induced idleness in fact implies $\forall i, \Delta_i = 0$, i.e., $\Delta_{\max} = 0$ for sporadic gang task sets. This is because if $\Delta_i > 0$ for some i, then the scenario where Δ_i was obtained must be possible to appear given the flexibility of sporadic releases, and schedulability analysis for sporadic gang tasks must cover all valid sporadic release patterns.

Therefore, the following corollary backs up the previous Claim 1 in Sec. 5, where "no parallelism-induced idleness" is interpreted more precisely by $\Delta_{\max} = 0$ as explained above.

Corollary 1. *For any task set such that $\Delta_{\max} = 0$, if $U_{sum} \leq M$, then each task τ_i must have tardiness at most $x + e_i$ where*

$$x = \max\left\{\frac{(M-1)e_{\max} - e_{\min}}{M \cdot (1 - \lambda_{\max}) + \lambda_{\max}}, 0\right\}.$$

Proof. This corollary directly follows by Theorem 1, taking $\Delta_{\max} = 0$ in both Eq. (10) and Eq. (11). □

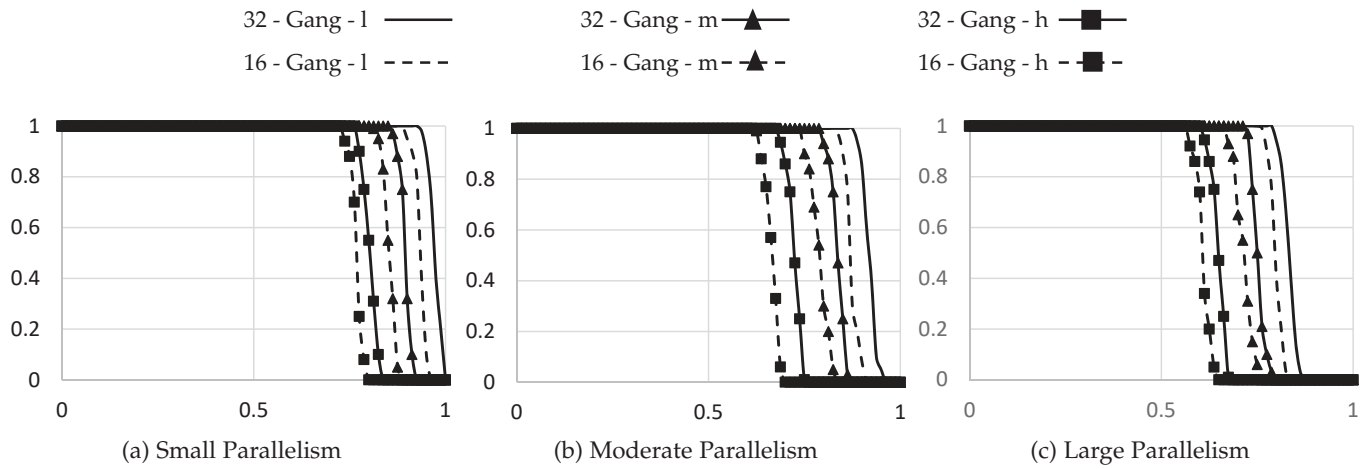
Meanwhile, we establish the following claim, which indicates that Eq. (10) as a utilization bound about M and $U_{\max}$ for *any* task set is tight while it might not be an exact SRT-schedulability test for a *specific* task set.

Claim 2. *For any M and $\Delta_{\max}$, there exists a system that is SRT-infeasible and its total utilization is greater than but arbitrarily close to $(M - \Delta_{\max})$.*

To show the above claim, we demonstrate that:

For any given integer values $M^* > \Delta^* > 0$, we can construct a system such that

1) There are exact M^* processing units in this system;
2) This system is SRT-infeasible, i.e., tardiness of some task increasing without bound is inevitable;
3) In this system, $\Delta_{\max} = \Delta^*$, where $\Delta_{\max} = \max_i\{\Delta_i\}$ and each Δ_i is as defined in Def. 4;



(a) Small Parallelism (b) Moderate Parallelism (c) Large Parallelism

Fig. 7: Schedulability results. In all graphs, the x-axis denotes the task set average utilization ($\frac{U_{sum}}{M}$) cap and the y-axis denotes the schedulability which is the fraction of generated task sets that were schedulable. In the first (respectively, second and third) column of graphs, small (respectively, moderate and large) per-task parallelism is assumed. Each graph gives three curves for the cases of light, medium, and heavy per-core utilization (e/p), respectively. As seen at the top of the figure, the label "16-Gang-l(m/h)" indicates the approach of our utilization-based test assuming light (medium/heavy) per-task utilization and $M = 16$. Similarly, "32-Gang" labels are used to denote $M = 32$ under three scenarios.

4) The total utilization of this system is greater than but arbitrarily close to $M^* - \Delta^*$.

Such task system construction is given by the following Example 8. Please note that, it is more intuitive to develop a construction as the one in Case 1 below, which satisfies the requirements 1, 2, and 4 above in all cases; however, when $\Delta^* < \frac{M^*-1}{2}$, this construction does not satisfy the requirement 3, because $\Delta_{\max}$ would be $(M^*-\Delta^*-1)$ instead of Δ^* in this case. Therefore, for the case of $\Delta^* < \frac{M^*-1}{2}$, we have to develop a different construction as presented in Case 2.

Example 8. *For any given value of M^*, we consider the scheduling of two sporadic gang tasks, namely τ_1 and τ_2, on exact M^* processing units. For any given value of Δ^*, we construct τ_1 and τ_2 by the following two cases. Fig. 6 provides a visual illustration for the task system in the two respective cases, while the actual schedule may vary as we are discussing* feasibility *which means* for any scheduling algorithm.

Case 1: $\Delta^* \geq \frac{M^*-1}{2}$. *In this case, we consider the two tasks: $\tau_1 = (\varepsilon, \Delta^* + 1, 10)$ and $\tau_2 = (10, M^* - \Delta^*, 10)$. Since $m_1 + m_2 = (\Delta^* + 1) + (M^* - \Delta^*) = M^* + 1 > M^*$, the two tasks can never execute in parallel with each other. Also, both tasks have a period of* 10 *but may have a combined execution time of $10 + \varepsilon > 10$ in every period. Thus, the tardiness of at least one of the two tasks will increase without bound in the worst case, no matter what scheduling algorithm is applied. At the same time, please note that $\Delta_1 = \Delta^*$ (when τ_1 is pending and τ_2 is scheduled) and $\Delta_2 = M^* - \Delta^* - 1$ (when τ_2 is pending and τ_1 is scheduled). Because $\Delta^* \geq \frac{M-1}{2}$ in this case, it follows that $\Delta^* \geq M^* - \Delta^* - 1$. Thus, it is true that $\Delta_{\max} = \Delta^*$ in this system. Also, the total utilization of this system is $(\frac{\Delta^*+1}{10} \cdot \varepsilon + M^* - \Delta^*)$, which is greater than but arbitrarily close to $(M^* - \Delta^*)$ when $\varepsilon \to 0^+$.*

Case 2: $\Delta^* < \frac{M^*-1}{2}$. *In this case, we consider the two tasks: $\tau_1 = (\varepsilon, M^* - \Delta^*, 10)$ and $\tau_2 = (10, M^* - \Delta^*, 10)$. Because $\Delta^* < \frac{M^*-1}{2}$, which implies $2\Delta^* < M^* - 1$, in Case 2, it follows that $m_1 + m_2 = 2M^* - 2\Delta^* > M^* + 1 > M^*$. Therefore, the two tasks can never execute in parallel with each other. For the same reason as that in Case 1 above, the tardiness of at least one of the two tasks will increase without bound in the worst case, no matter what scheduling algorithm is applied. At the same time, it is clear that $\Delta_1 = \Delta_2 = \Delta^*$. Thus, it is true that $\Delta_{\max} = \Delta^*$. The total utilization of this system is $(\frac{M^*-\Delta^*}{10} \cdot \varepsilon + M^* - \Delta^*)$, which is greater than but arbitrarily close to $(M^* - \Delta^*)$ when $\varepsilon \to 0^+$.*

A key observation from Example 8 is that there always exists at least one task system that exceeds the utilization constraint stated in Eq. 10 by an arbitrarily small positive real number, having unbounded tardiness. This observation implies that our derived schedulability test is unlikely to be improved without incorporating further information and investigation beyond M and $\Delta_{\max}$.

8 Evaluation

In this section, we describe experiments conducted to evaluate the applicability of the schedulability tests proposed in this work. Our goal is to examine how restrictive the derived schedulability tests' utilization caps are. Specifically, we evaluated our derived utilization-based test given by Eq. 10.

Experimental setup. In our experiments, parameters of the gang tasks are generated as follows: task periods were uniformly distributed over $[20ms, 200ms]$; per-core utilization (i.e., $\frac{e_i}{p_i}$) for different task sets was distributed differently for each experiment using three uniform distributions. The ranges for the uniform distributions were $[0.005, 0.1]$ (light), $[0.1, 0.3]$ (medium), and $[0.3, 0.8]$ (heavy). Parallelism of tasks were also distributed using three uniform distributions: $[1, \frac{M}{4}]$ (parallelism is small), $[\frac{M}{4}, \frac{5M}{8}]$ (parallelism is moderate), and $[\frac{5M}{8}, \frac{7M}{8}]$ (parallelism is high), where M denotes the number of processors. Task execution costs were calculated from periods, utilization, and parallelism. The average total system utilization $\frac{U_{sum}}{M}$ are varied within $\{0.1, 0.2, \ldots, 1\}$. For each combination of per-core utilization, parallelism, and total utilization, 10,000 task sets were

generated for multicore platforms with $M = 16$ and $M = 32$ processors. Each such task set was generated by creating tasks until total utilization exceeds the corresponding utilization cap, and by then reducing the last task's utilization so that the total utilization equalled the utilization cap. For each generated system, SRT schedulability was checked for the proposed test. Note that similar task set generation methods have been used in some previous works [17], [3], [5].

Schedulability results. The obtained schedulability results are shown in Fig. 7 (the organization of which is explained in the figure's caption). Each curve plots the fraction of the generated task sets successfully scheduled by the corresponding approach, as a function of tasks' total utilization. As seen in Fig. 7, our proposed test is able to yield reasonably good performance in almost all cases, particularly when the parallelism is small. For instance, in Figure 7a, when tasks' parallelism is small and per-core utilization is light, more than 90% of the generated task sets are schedulable under both 32-Gang and 16-Gang. Another interesting observation is, in all tested scenarios, 32-Gang outperforms 16-Gang by a notable margin. For example, as seen in Fig. 7b, when tasks' parallelism is moderate and per-core utilization is medium, 32-Gang can achieve 100% schedulability when average U_{sum} equals 0.7875 while 16-Gang can only guarantee 100% schedulability when average U_{sum} is not greater than 0.7125. The intuition behind this observation is that as more processors are involved in the system, the parallelism of the multiprocessor platform can be better explored by the gang tasks. We also observe that both of the two cases perform best under small parallelism and light per-core utilization. This is because when parallelism is small and per-core utilization is light, $\Delta_{\max}$ become small, which clearly helps both two cases achieve higher schedulability. Another interesting observation is that, in each scenario, for both $M = 16$ and $M = 32$, the schedulability of gang tasks decreases when the per-core utilization increases. This is because when per-core utilization is large, the number of gang tasks generated for each task set becomes small under the corresponding utilization cap. According to the algorithm given in the appendix of [5], Δ_i is calculated based on the parallelisms of gang tasks from a selected subset of the gang task system. When the number of gang tasks becomes small, the possible combinations of the tasks from the subset becomes small, which yields a large $\Delta_{\max}$. According to our proposed schedulability test, with a large $\Delta_{\max}$, the utilization loss of the multicore platform is large.

9 Conclusion

In this paper, we have shown that gang tasks' non-malleable parallelism introduces a significant challenge to the derivation soft real-time schedulability analysis. We found that the multicore platform cannot be fully utilized by the gang tasks due to parallelism-induced idleness. Based on this key observation, we established and proved the first tardiness bounds for gang-scheduled sporadic task systems under preemptive GEDF scheduling in two steps: (*i*) upper bound the parallelism-induced idleness on the schedule using a dynamic programming method; (*ii*) apply the classic lag-based reasoning to calculate the gang tasks' tardiness bounds. Our derived schedulability test is unlikely to be improved without incorporating further information and investigation beyond the platform capacity and maximum parallelism-induced idleness. In future work, we plan to consider deriving tardiness bounds for a broader class of schedulers and for schedulers that must function in settings where processors are federated [15]. We also plan to investigate whether tighter tardiness bounds can be obtained by investigating more task parameters in the analysis.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive and helpful feedback. This work is supported by the U.S. National Science Foundation under Grant Nos. CNS-2038727, CNS-1750263, CNS-1618185 and IIS-1724227, start-up and REP grants from Texas State University and a start-up grant from Wayne State University.

References

[1] Vandy Berten, Pierre Courbin, and Joël Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *5th junior researcher workshop on real-time computing*, pages 9–12, 2011.

[2] Ashikahmed Bhuiyan, Kecheng Yang, Samsil Arefin, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. Mixed-criticality multicore scheduling of real-time gang task systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019.

[3] UmaMaheswari C Devi and James H Anderson. Flexible tardiness bounds for sporadic real-time task systems on multiprocessors. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[4] UmaMaheswari C Devi and James H Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

[5] Zheng Dong and Cong Liu. Analysis techniques for supporting hard real-time sporadic gang task systems. *Real-Time Systems*.

[6] Jeremy P Erickson, James H Anderson, and Bryan C Ward. Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling. *Real-Time Systems*, 50(1):5–47, 2014.

[7] Francesco Flammini, Riccardo Naddei, Concetta Pragliola, and Giovanni Smarra. Towards automated drone surveillance in railways: State-of-the-art and future directions. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 336–348. Springer, 2016.

[8] Joël Goossens and Pascal Richard. Optimal scheduling of periodic gang tasks. *Leibniz transactions on embedded systems*, 3(1):04–1, 2016.

[9] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[10] Shinpei Kato and Yutaka Ishikawa. Gang edf scheduling of parallel task systems. In *2009 30th IEEE Real-Time Systems Symposium*.

[11] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE micro*, 31(5):7–17, 2011.

[12] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[14] Hennadiy Leontyev and James H Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71, 2010.

[15] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th ECRTS*.

[16] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[17] Cong Liu and James H Anderson. An o (m) analysis technique for supporting real-time self-suspending task systems. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 373–382. IEEE, 2012.
[18] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.
[19] Jinghao Sun, Nan Guan, Yang Wang, Qingqing He, and Wang Yi. Scheduling and analysis of realtime openmp task systems with tied tasks. In *Proceedings of Real-Time Systems Symposium*, 2017.
[20] Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H Anderson, F Donelson Smith, and Shige Wang. Making openvx really" real time". In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 80–93. IEEE, 2018.

Zheng Dong received the BS degree from Wuhan University, China, in 2007, the MS degree from University of Science and Technology of China, in 2011, and the PhD degree from the University of Texas at Dallas, USA, in 2019. He is an assistant professor with the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests are in real-time and embedded computer systems and mobile edge computing. His current research focus is on multiprocessor scheduling theory and hardware-software co-design for real-time applications. He received the Outstanding Paper Award at the 38th IEEE RTSS. He is a member of the IEEE Computer Society.

Kecheng Yang is an Assistant Professor in the Department of Computer Science at Texas State University. He received his Ph.D. and M.S. degrees in Computer Science from the University of North Carolina at Chapel Hill in 2018 and 2015, respectively, both with Prof. James H. Anderson. Before that, he received his B.E. degree in Computer Science and Technology from Hunan University in 2013. His research interests include real-time and cyber-physical systems, scheduling theory and resource allocation algorithms, and heterogeneous multiprocessor platforms.

Nathan Fisher received the BS degree from the University of Minnesota, Minneapolis, in 1999, the MS degree from Columbia University, New York, in 2002, and the PhD degree from the University of North Carolina, Chapel Hill, in 2007, all in computer science. He is a professor with the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests are in schedulability analysis, resource allocation and optimization, cache analysis for real-time systems, and approximation algorithms. He is a member of the IEEE Computer Society and currently serves as Treasurer for the IEEE Technical Committee on Real-Time Systems.

Cong Liu received the Ph.D. degree in computer science from the University of North Carolina at Chapel Hill, in Jul. 2013. He is an associate professor in the Department of Computer Science, University of Texas at Dallas. His research interests include real-time systems and GPGPU. He has published more than 30 papers in premier conferences and journals. He received the Best Paper Award at the 30th IEEE RTSS and the 17th RTCSA. He is a member of the IEEE.