Improving the Schedulability and Quality of Service for Federated Scheduling of Parallel Mixed-Criticality Tasks on Multiprocessors

Risat Mahmud Pathan
Chalmers University of Technology, Sweden
risat@chalmers.se

Abstract

This paper presents federated scheduling algorithm, called MCFQ, for a set of parallel mixed-criticality tasks on multiprocessors. The main feature of MCFQ algorithm is that *different* alternatives to assign each high-utilization, high-critical task to the processors are computed. Given the different alternatives, we carefully select one alternative for each such task so that *all* the other tasks can be successfully assigned on the remaining processors. Such flexibility in choosing the right alternative has two benefits. First, it has higher likelihood to satisfy the total resource requirement of all the tasks while ensuring schedulability. Second, computational slack becomes available by intelligently selecting the alternative such that the total resource requirement of all the tasks is minimized. Such slack then can be used to improve the QoS of the system (i.e., never discard some low-critical tasks). Our experimental results using randomly-generated parallel mixed-critical tasksets show that MCFQ can schedule much higher number of tasksets and can improve the QoS of the system significantly in comparison to the state of the art.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases mixed-criticality systems, real-time systems, multiprocessor scheduling, federated scheduling

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.12

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

1 Introduction

Multicore processors offer high computing power to meet the increasing demand of more advanced functions in many real-time systems like automotive and avionics. Multicores also provide the opportunity to integrate multiple functions having different levels of criticality on the same platform. The real-time tasks of such mixed-critical (MC) systems require different levels of assurance in meeting their deadlines. A relatively high-critical task requires a higher level of assurance in meeting its deadline because such a task is often safety-critical and its correctness under very pessimistic assumptions needs to be approved by the certification authority (CA). On the other hand, the system designers' objective is to ensure the correctness of both high- and low-critical tasks but under relatively less pessimistic assumptions. The different concerns and pessimism between the CA and system designer makes it challenging to develop a real-time multiprocessor scheduling strategy for MC system.

Parallel programming paradigm allows both inter- and intra-task parallelism to effectively exploit the processing capacity of a parallel multicore architecture: each real-time task can be implemented using a task-based parallel programming model such as OpenMP4.0 [33], where the dependencies between sequential chunks of computation (called, subtasks) are specified

30th Euromicro Conference on Real-Time Systems (ECRTS 2018).
Editor: Sebastian Altmeyer; Article No. 12; pp. 12:1–12:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

by programmers. Thus, each parallel task can be viewed as a direct acyclic graph (DAG), where the nodes are subtasks and edges are dependencies (called, precedence constraints) between the subtasks. This paper presents a scheduling algorithm and its analysis for a collection of dual-criticality sporadic DAG tasks on multiprocessors where each task is either a high-critical (HI) task or a low-critical (LO) task.

Several works on scheduling (non-MC) sporadic DAG tasks on multiprocessors [25, 31] abstract the complex internal structure of each DAG task using only two parameters: *total work* and *critical-path length*. The total work of a task τ_i is the sum of the worst-case execution times (WCETs) of all the subtasks of task τ_i. The critical-path length of task τ_i is the maximum sum of the WCETs of the subtasks that belong to any source-to-sink path of task τ_i. Li et al. [27] proposed mixed-critical DAG task model by associating a *nominal* and an *overload* value for the total work and critical-path length for each DAG task. The nominal and overload total work of a DAG task τ_i are respectively denoted by C_i^N and C_i^O such that $C_i^N \leq C_i^O$. Similarly, the nominal and overload critical-path length are respectively denoted by L_i^N and L_i^O where $L_i^N \leq L_i^O$. Li et al. [26] recently (September, 2017) proposed federated scheduling of implicit-deadline MC sporadic DAG tasks, called MCFS-Improve, which is an improvement of their original work in [27].

The basic idea of federated scheduling is the following. Each MC task τ_i with overload utilization larger than 1 is assigned to a set of dedicated processors and all the low-utilization tasks are assigned on the remaining processors. Each task is also assigned a virtual deadline [4] such that the task meets its deadline if it does not overrun its nominal total work and critical-path length. The runtime system has two states: *typical* and *critical*. Each task initially starts in typical state. The state of the system is switched from typical to the critical state when some job does not signal completion by its virtual deadline. During the critical state, the LO-critical tasks may need to be discarded to allocate additional computing resource to the HI-critical tasks so that each HI-critical task meets its deadlines during the critical state. Li et al. [27, 26] proposed a very interesting algorithm to assign a collection of MC sporadic DAG tasks to a given number of processors and apply a schedulability test to determine whether the assignment guarantees the MC-correctness of the system or not (the formal definition of MC-correctness will be presented shortly).

By carefully analyzing the task-assignment algorithm in [26], we observed that the number of dedicated processors required for individual high-utilization task does not take into account how many processors are required for the other tasks. Consequently, the task assignment in MCFS-Improve [26] may declare failure due to not having enough number of processors for all the tasks even if there exists another way of allocating dedicated processors to individual high-utilization task. Our second observation is that the task assignment algorithm in MCFS-Improve does not explicitly consider to maximize the number of LO-critical tasks that do not need to be discarded in the critical state. Maximizing the number of LO-critical tasks that are never discarded is important to improve the QoS of the system.

The task assignment algorithm is very crucial for guaranteeing the MC-correctness for federated scheduling on multiprocessors. Since the problem of assigning tasks to the processors (even for sequential tasks) is NP-hard in the strong sense, designing an effective task assignment algorithm for federated scheduling is not only important but also more challenging for parallel tasks in comparison to sequential tasks. To this end, we propose a new task assignment algorithm for federated scheduling, called Mixed-Criticality Federated Scheduling with QoS (MCFQ), and empirically show that the performance is significantly better in terms of both schedulability and improving the QoS of the system in comparison to MCFS-Improve.

The main feature of MCFQ algorithm is that it finds *different* alternative ways to assign individual high-utilization task to different number of dedicated processors based on a new schedulability test. After all the different alternatives to assign each high-utilization task are computed, we carefully select one particular alternative for each high-utilization task such that *all* the tasks can be successfully assigned to the available number of processors. In contrast to the task-assignment algorithm in [27, 26] that makes "local" decision about task assignment when analyzing each individual high-utilization task separately, we make a "global" decision by taking into account how processors can be intelligently allocated to the tasks so that there are enough processors for all the tasks. The main contributions of this paper are the following:

- A new federated scheduling algorithm MCFQ for a set of implicit-deadline MC sporadic DAG tasks on M processors is proposed. A new schedulability analysis for the high-utilization and HI-critical tasks is proposed. The main outcome of the analysis is a polynomial-time schedulability test that can be used to determine different alternatives for allocating such tasks to dedicated processors. Based on the different alternatives for assigning the high-utilization and HI-critical tasks, we ultimately find different alternatives to assign *all* the tasks to the processors such that MC-correctness for each such alternative is guaranteed.
- We select the alternative to assign all the tasks that minimizes the total number of processors required during the critical state, which maximizes the number of *unused* processor during the critical state. The unused processors during the critical state are used to meet the demand of additional computing capacity of the HI-critical tasks rather than discarding some or all the LO-critical tasks. We apply Integer Linear Programming (ILP) to maximize the number of such non-discarding LO-critical tasks.
- Empirical investigation using randomly-generated tasksets shows that both the number of schedulable tasksets and the QoS of the system using MCFQ algorithm are significantly higher than the state-of-the-art MCFS-Improve algorithm.

The remainder of this paper is organized as follows. Section 2 presents the system model and useful definitions that are used in this paper. An overview of the MCFQ algorithm is presented in Section 3. The detailed schedulability analysis of the MCFQ algorithm is presented in Section 4. Empirical investigation is presented in Section 5. Finally, related works are presented in Section 6 before concluding in Section 7.

2 System Model and Useful Definitions

We consider scheduling a set $\Gamma = \{\tau_1, \ldots \tau_n\}$ of n implicit-deadline MC sporadic DAG tasks on M identical processors such that each processor has a (normalized) speed of one. Each task τ_i is characterized by the tuple $(Z_i, T_i, D_i, C_i^N, C_i^O, L_i^N, L_i^O)$ where

- $Z_i \in \{\texttt{HI}, \texttt{LO}\}$ is the criticality of the task: LO and HI specifies that task τ_i is a low-critical task and a high-critical task, respectively;
- $T_i \in \mathbb{R}^+$ is the minimum inter-arrival time of the jobs (i.e., called the period) of the task;
- $D_i \in \mathbb{R}^+$ is the relative deadline the task such that $D_i = T_i$;
- C_i^N and C_i^O are the maximum nominal and maximum overload total work for any job of task τ_i where $C_i^N \leq C_i^O$ for $Z_i = \texttt{HI}$ and $C_i^O = C_i^N$ for $Z_i = \texttt{LO}$; and
- L_i^N and L_i^O are the maximum nominal and maximum overload critical-path length for any job of task τ_i where $L_i^N \leq L_i^O$ for $Z_i = \texttt{HI}$ and $L_i^O = L_i^N$ for $Z_i = \texttt{LO}$.

If a job of task τ_i is released at time r, then it must complete its execution by time $(r+D_i)$. The nominal and overload utilizations of task τ_i are respectively denoted by u_i^N and u_i^O such that $u_i^N = C_i^N/D_i$ and $u_i^O = C_i^O/D_i$. If $u_i^O > 1$, then task τ_i is a *high-utilization* task; otherwise, it is a *low-utilization* task. Based on the overload utilization and the criticality, the tasks in set Γ are categorized in four *disjoint* subsets $\Gamma_{\mathtt{HH}}$, $\Gamma_{\mathtt{HL}}$, $\Gamma_{\mathtt{LH}}$, and $\Gamma_{\mathtt{LL}}$ as follows:

$$\Gamma_{\mathtt{HH}} = \{\tau_i \mid u_i^O > 1 \text{ and } Z_i = \mathtt{HI}\} \qquad \Gamma_{\mathtt{LH}} = \{\tau_i \mid u_i^O > 1 \text{ and } Z_i = \mathtt{LO}\}$$
$$\Gamma_{\mathtt{HL}} = \{\tau_i \mid u_i^O \leq 1 \text{ and } Z_i = \mathtt{HI}\} \qquad \Gamma_{\mathtt{LL}} = \{\tau_i \mid u_i^O \leq 1 \text{ and } Z_i = \mathtt{LO}\}$$

Note that $\Gamma = \Gamma_{\mathtt{HH}} \cup \Gamma_{\mathtt{LH}} \cup \Gamma_{\mathtt{HL}} \cup \Gamma_{\mathtt{LL}}$. We will use the following lemmas later in this paper.

▶ **Lemma 1.** *Consider a* `MC` *DAG task τ_i. The following property is satisfied:*

$$(C_i^O - C_i^N) \geq (L_i^O - L_i^N) \tag{1}$$

Proof. By the definition of total work and critical-path length, it is evident that the total work includes the work on the critical path. Therefore, the difference between the overload and nominal total work is larger than or equal to the difference between the overload critical-path length and nominal critical-path length. Therefore, $(C_i^O - C_i^N) \geq (L_i^O - L_i^N)$. ◀

▶ **Lemma 2.** *Consider a job J of a DAG task τ that is released at time r and executes on m dedicated processors using a work-conserving algorithm*[1] *where $m \geq 1$. If the remaining total work and the remaining length of the critical path at time $(r+t)$ are respectively C and L where $t \geq 0$, then job J completes its execution no later than at time $(r+R)$ such that*

$$R \leq t + L + \frac{C - L}{m} \tag{2}$$

Proof. Since $(r+R)$ is the time at which the job completes its execution, there is at least one processor busy executing the nodes of job J in the interval $[r+t, r+R]$. Let ℓ is the cumulative length of intervals in $[r+t, r+R]$ during which at least one processor is idle where $\ell \geq 0$. Therefore, all the m processors are simultaneously busy for a cumulative length of intervals equal to $(R - t - \ell)$ in the interval $[r+t, r+R]$.

Since the remaining length of the critical path decreases when there is at least one processor idle, we have $0 \leq \ell \leq L$. Therefore, the total work completed during the interval $[r+t, r+R]$ is at least $\ell + m \cdot (R - t - \ell)$. Since the the maximum remaining total work at time $(r+t)$ is C, we have

$$\begin{aligned}
& \ell + m \cdot (R - t - \ell) \leq C \\
& (\text{since } m \geq 1 \text{ and } \ell \leq L) \\
\Rightarrow\ & L + m \cdot (R - t - L) \leq C \\
\Leftrightarrow\ & R \leq t + L + \frac{C - L}{m}
\end{aligned}$$
◀

Our proposed `MCFQ` scheduling algorithm assigns a virtual deadline, denoted by D_i^v, to each task τ_i. As will be evident later, the virtual deadline for each task is assigned such that each job of task τ_i is guaranteed to meet its deadline by it virtual deadline if the total work and critical-path length does not exceed their nominal values C_i^N and L_i^N, respectively.

[1] A work conserving algorithm is any scheduling algorithm that never idles a processor if there is a node waiting for execution.

States. The system operates either in typical or critical state. The system starts in typical state. If each job of each task τ_i signals completion by its virtual deadline D_i^v, then the system remains in the *typical state*. If any job does not complete by its virtual deadline (i.e., either the total work or critical-path length exceeds the nominal value), then the system is said to *switch* from typical to critical state. Once the system switches to the critical state, jobs of the LO-critical tasks may be discarded. The system remains in the critical state if each job of the HI-critical task signals completion without overrunning its overload total work C_i^O and overload critical-path length L_i^O. All other states are erroneous.

Correctness. We define an algorithm for scheduling a set of MC tasks to be correct if the following properties are satisfied:

- During the typical state, all the jobs of each task meet their deadlines.
- During the critical state, all the jobs of each HI-critical task meets their deadlines.

It is evident from the definition of correctness that if the state of the system is changed from typical to critical, then the runtime scheduler can discard the execution of such LO-critical tasks during its critical state in order to provide additional computing resource to ensure the correctness of the HI-critical tasks. The system can switch back from critical to typical state based on the approach proposed by Li et al. [26, p. 794].

3 An Overview of the MCFQ Algorithm

The MCFQ scheduling works in two phases: an offline task assignment phase and an online runtime scheduling phase. In this section, we present an overview of the task-assignment phase and the runtime scheduler of MCFQ. In Section 4, we present the details of the task-assignment phase, present the schedulability analysis, and prove the correctness of MCFQ.

Task Assignment Phase. This phase determines the mapping of the tasks to the processors and also computes a virtual deadline D_i^v for each task τ_i. The idea of virtual deadline, originally proposed for EDV-VD scheduling of sequential tasks [4], is also used by Li et al. [27, 26] for MCFS-Improve algorithm. The virtual deadline of each task τ_i is used by the runtime scheduler to determine whether the system needs to switch from typical to critical state or not. The method to compute the virtual deadline will be presented shortly.

The MCFQ scheduling algorithm assigns each task to the processors based on whether it is a high- or low-utilization task. It assigns each high-utilization (i.e., $u_i^O > 1$) task $\tau_i \in (\Gamma_{HH} \cup \Gamma_{LH})$ to a set of dedicated processors. We denote π_i^N and π_i^O the number of dedicated processors assigned to a high-utilization task τ_i for the typical and critical states, respectively. The number of dedicated processors assigned to each HH task $\tau_i \in \Gamma_{HH}$ for the typical and critical states satisfies $\pi_i^N \leq \pi_i^O$. A HH task τ_i is assigned additional $(\pi_i^O - \pi_i^N)$ processors to guarantee its correctness only if τ_i does not complete by its virtual deadline.

For each LH task τ_i, the task-assignment only determines the number of dedicated processors π_i^N for the typical state. A LH task τ_i may need to provide its π_i^N processors (by discarding τ_i) to some HI-critical task τ_k during the critical state. Therefore, π_i^O=0 for each LH task τ_i if such a task is dropped; otherwise, π_i^O=π_i^N to specify that τ_i is never dropped.

We assign the low-utilization tasks in set $\Gamma_{HL} \cup \Gamma_{LL}$. Since $u_i^O \leq 1$ for each task $\tau_i \in (\Gamma_{HL} \cup \Gamma_{LL})$, we have $C_i^N \leq C_i^O \leq D_i$. Therefore, such a low-utilization task τ_i can execute sequentially and does not necessarily require parallelism to meet its deadline. Similar to [27, 26], the MCFQ algorithm also assigns all the low-utilization tasks using the MC-Partition-0.75 algorithm proposed in [6]. By applying MC-Partition-0.75 algorithm [6] on all the low-utilization tasks, we determine the *minimum* number of processors required to ensure the correctness of these low-utilization tasks. Note that MC-Partition-0.75 algorithm allocates for all the low-utilization tasks the same number of processors for both the typical and critical states. Let $\Pi_{\texttt{LU}}$ is the minimum number of processors (computed by applying MC-Partition-0.75 algorithm) required for the correctness of all the low-utilization tasks during the typical and critical states. The minimum can be found by applying a bisection search.

After the number of processors determined for each high-utilization task and for all the low-utilization tasks for the typical and critical states is determined, we apply the **capacity constraint**: *if the total number of processors required by all the tasks during each individual state is not more than M (i.e., number of available processors), then the task-assignment phase declares success; otherwise, it declares failure.*

Before the task assignment phase starts, we assume that all the LH tasks may need to be dropped (i.e., we assume $\pi_i^O = 0$ for each $\tau_i \in \Gamma_{\texttt{LH}}$). After the MCFQ algorithm finds a successful assignment for all the tasks by assuming that all LH tasks are dropped, it may be the case that the total number of processors required during the critical state for all the tasks is smaller than M. In other words, there may be (unused) processors that have no task assigned during the critical state. If the number of such unused processors during the critical state is more than π_i^N for some LH task τ_i, then we set $\pi_i^O = \pi_i^N$ to specify that such a LO-critical task is never dropped. Such adjustment will not compromise the schedulability of the HH task τ_k because the additional $(\pi_k^O - \pi_k^N)$ processors to the HH task τ_k during the critical state can be assigned from the set of idle processors rather than discarding the LH task τ_i during the critical state.

Run-Time Scheduler. The runtime scheduler of MCFQ algorithm works as follows:

- The system starts in typical state. During the typical state,
  - the nodes of each high-utilization task τ_i are scheduled using any work conserving scheduling algorithm on π_i^N number of dedicated processors; and
  - the nodes of all the low-utilization tasks are scheduled on $\Pi_{\texttt{LU}}$ processors on which they are assigned by the MC-Partition-0.75 algorithm.
- If any HI-critical task τ_i does not signal completion by its virtual deadline D_i^v, then the system switches from typical to the critical state, and
  - If $u_i^O > 1$, then one by one active (i.e., not dropped yet) LH task τ_k for which $\pi_k^O = 0$ is dropped until additional $(\pi_i^O - \pi_i^N)$ processors to the HH task τ_i are assigned. The nodes of the HH task τ_i are now scheduled using a work conserving scheduling algorithm on π_i^O dedicated processors.
  - If $u_i^O \leq 1$, the all the LL tasks are dropped and the HL tasks are scheduled on the $\Pi_{\texttt{LU}}$ processors on which they are assigned by the MC-Partition-0.75 algorithm.

Note that if some HL task τ_i (i.e., $u_i^O \leq 1$) triggers the switching of system's state from typical to critical, then all the LL tasks are dropped (no LH task is dropped) since all the HL tasks (according to [6]) still meets their deadline on $\Pi_{\texttt{LU}}$ processors during the critical state. If some HH task τ_i (i.e., $u_i^O > 1$) triggers the switching of system's state from typical to critical, then adequate number of LH tasks are dropped to assign the HH task τ_i additional $(\pi_i^O - \pi_i^N)$ processors. The remaining (not yet dropped) LH tasks may continue execution until some other HH task does not complete by its virtual deadline. Therefore, the system may degrade gracefully as is pointed by Li et al.in [27, 26].

Practicality of Federated Scheduling. The practical consideration of federated scheduling of parallel DAG tasks is discussed in [25] by pointing out that there is no preemption on any high-utilization task since each such task has a dedicated number of processors. Note that a low priority parallel task in global scheduling (all processors are shared) or partitioned scheduling (more than one task may share a dedicated subset of the processors) may suffer from preemption. Li et al. in [26] also developed a reference system written in OpenMP by implementing the `MCFS-Improve` scheduling in Linux using the RT_PREEMPT patch as the underlying RTOS. It has been experimentally shown in [26] that the overhead of real implementation of federated scheduling for parallel `MC` tasks is low. Since the runtime scheduling of `MCFS-Improve` and our proposed `MCFQ` algorithms are fundamentally the same, the `MCFQ` algorithm can also be implemented the same way as in [26] and is also expected to have very low implementation overhead.

4 Schedulability Analysis and Task Assignment of MCFQ Algorithm

This section presents the task assignment strategy of `MCFQ` algorithm. The schedulability analysis of the tasks in sets $\Gamma_{\texttt{LH}}$ and $\Gamma_{\texttt{HH}}$ are presented in subsections 4.1 and 4.2, respectively. Recall that the MC-Partition-0.75 is used to determine the minimum number of processors $\Pi_{\texttt{LU}}$ to correctly schedule the tasks in set $(\texttt{HL} \cup \texttt{LL})$. The total number of processors required to guarantee the correctness for all the tasks in Γ is determined in subsection 4.3.

4.1 Task Assignment: LH tasks

In this section, the number of processors π_i^N required to ensure the correctness of `LH` task τ_i is determined. The virtual deadline for each `LH` task τ_i is $D_i^v = D_i$. For the time being, we assume π_i^O=0 for each `LH` task τ_i (such a task is dropped during the critical state). In subsection 4.4, we will determine which `LH` tasks do not need to be dropped and we reset $\pi_i^O = \pi_i^N$ for such `LH` tasks.

▶ **Lemma 3.** *The execution of each LH task $\tau_i \in \Gamma_{LH}$ is correct using the runtime scheduler of MCFQ algorithm if task τ_i is assigned π_i^N dedicated processors during the typical state where*

$$\pi_i^N = \lceil (C_i^N - L_i^N)/(D_i - L_i^N) \rceil \tag{3}$$

Proof. The proof is same as the proof in [26, (Lemma 2, p. 771)]. ◀

4.2 Task Assignment: HH Tasks

In this subsection, the schedulability analysis of each `HH` task τ_i in order to determine the number of processors required to ensure its correctness during the typical and critical states is presented. Each `HH` task τ_i is also assigned a virtual deadline D_i^v.

The outcome of the analysis is a schedulability test, denoted by $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$, where μ_i^N and μ_i^O are respectively the number of dedicated processors assigned to task τ_i during the typical and critical states such that $1 \leq \mu_i^N \leq \mu_i^O$. If the schedulability test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ is satisfied, then it is guaranteed that task τ_i meets its deadline where $\mu_i^N$and $\mu_i^O$are the number of dedicated processors for τ_i during the typical and critical state, respectively.

Since there are M processors on the multiprocessor platform, we apply $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ for all possible pairs of (μ_i^N, μ_i^O) where $\mu_i^N = 1, 2, \ldots M$ and $\mu_i^O = \mu_i^N, \mu_i^N + 1, \ldots M$ to determine the valid pairs of (μ_i^N, μ_i^O) for which `HH` task τ_i meets its deadline during the typical and critical states. From all the valid pairs (μ_i^N, μ_i^O) for each `HH` task $\tau_i \in \Gamma_{\texttt{HH}}$, we select one pair for each `HH` task τ_i as the final values of π_i^N and π_i^O. The opportunity to select the values of π_i^N and π_i^O from the different possible pairs has higher likelihood of satisfying the capacity constraints of the platform, which is demonstrated using the following example.

▶ **Example 4.** Consider a multiprocessor platform $M = 16$ and a taskset with three high-utilization MC tasks. There is one LH task τ_a and two HH tasks τ_b and τ_c. The specific values of the total work and critical-path length of these tasks are not needed to understand this example. Assume that $\pi_a^N = 5$ for the LH task τ_a and $\pi_a^O = 0$. Also assume that the SCHH(τ_b, μ_b^N, μ_b^O) test is satisfied for only one pair $(\mu_b^N, \mu_b^O) = (4, 9)$ for task τ_b. Since there is only one pair $(\mu_b^N, \mu_b^O) = (4, 9)$ for HH task τ_b, we have only one option for selecting the final values of π_b^N and π_b^O such that $\pi_b^N = \mu_b^N = 4$ and $\pi_b^O = \mu_b^O = 9$.

Finally, consider that SCHH(τ_c, μ_c^N, μ_c^O) is satisfied for two different pairs $(\mu_c^N, \mu_c^O) = (5, 8)$ and $(\mu_c^N, \mu_c^O) = (6, 7)$ for task τ_c. Since there are two possible pairs of (μ_c^N, μ_c^O) for task τ_c, there are two possible ways to select the final values of π_c^N and π_c^O for task τ_c.

If we select $(\pi_c^N, \pi_c^O) = (\mu_c^N, \mu_c^O) = (5, 8)$ for task τ_c, the total number of processors for the three tasks τ_a, τ_b and τ_c during the typical state is $(5 + 4 + 5) = 14$, which is not larger than $M = 16$. The total number of processors for the three tasks τ_a, τ_b and $\tau_c$during the critical state is $(0 + 9 + 8) = 17$, which is *larger* than $M = 16$. Consequently, the capacity constraint is *not* satisfied and the overall task allocation phase declares failure.

If we select $(\pi_c^N, \pi_c^O) = (\mu_c^N, \mu_c^O) = (6, 7)$ for task τ_c, the total number of processors for the three tasks τ_a, τ_b and τ_c during the typical state is $(5 + 4 + 6) = 15$, which is not larger than $M = 16$. The total number of processors for all the three tasks during the critical state is $(0 + 9 + 7) = 16$, which is not larger than $M = 16$. Consequently, the capacity constraint is satisfied for both states and the overall task allocation phase declares success. ◀

Example 4 demonstrates that the selection of the final values of π_i^N and π_i^O for each of the HH tasks τ_i from the different alternative pairs of (μ_i^N, μ_i^O) is crucial to the overall success of the task assignment algorithm of MCFQ. Before we present the schedulability test SCHH(τ_i, μ_i^N, μ_i^O) in Lemma 5, we present how virtual deadline D_i^v is assigned to τ_i.

Virtual Deadline Assignment. Consider that the number of dedicated processors for HH task τ_i during the typical and critical states are μ_i^N and μ_i^O, respectively. The virtual deadline D_i^v for HH task τ_i is assigned as follows:

$$D_i^v = L_i^N + (C_i^N - L_i^N)/\mu_i^N \tag{4}$$

▶ **Lemma 5** (Schedulability test SCHH(τ_i, μ_i^N, μ_i^O)). *Consider a pair (μ_i^N, μ_i^O) such that the HH task τ_i is assigned μ_i^N and μ_i^O dedicated processors respectively for the typical and critical states where $1 \le \mu_i^N \le \mu_i^O$. Each job of task τ_i meets its deadline in all the correct states if the following equation is satisfied:*

$$D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O + min\{L_i^N, \frac{\omega_i}{\mu_i^N}\} \cdot (1 - \frac{\mu_i^N}{\mu_i^O}) \tag{5}$$

where $\omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)$.

Proof. Since $(C_i^O - C_i^N) - (L_i^O - L_i^N) \ge 0$ from Eq. (1), we have $\omega_i \ge 0$. Moreover, $L_i^N \ge 0$ and $\mu_i^N \ge 1$. It follows that $min\{L_i^N, \omega_i/\mu_i^N\} \ge 0$. Because $\mu_i^O \ge \mu_i^N$, we also have $(1 - \mu_i^N/\mu_i^O) \ge 0$. Therefore, $min\{L_i^N, \omega_i/\mu_i^N\} \cdot (1 - \mu_i^N/\mu_i^O) \ge 0$ and from Eq (5) we have

$$D_i \ge \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O \tag{6}$$

Consider a generic job J_i of task τ_i. Without loss of generality assume that the job is released at time 0. The entire execution of job J_i happens in any of the three possible scenarios: (i) stable typical state, (ii) stable critical state, and (iii) during the transition from typical to critical state. A stable state refers to the situation when there is no switching of states during the execution of job J_i. This lemma is proved by showing that job J_i meets its deadline for all these three scenarios if Eq (5) is satisfied. Since we are considering implicit-deadline tasks, if the generic job J_i meets its deadline by time D_i, then each other job of τ_i will also meet its deadline.

Stable typical state. During the stable typical state, the subtasks of task τ_i are executed using any work-conserving scheduling algorithm on μ_i^N dedicated processors. Since job J_i executes entirely in stable typical state, it signals completion at or before its virtual deadline D_i^v. We will show that $D_i^v \leq D_i$, which shows job J_i meets its deadline during the stable typical state. Since $L_i^O \geq L_i^N$, from Eq. (6) we have $D_i \geq \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^N$. Since $\frac{\omega_i}{\mu_i^O} \geq 0$ and $D_i^v = \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N$ from Eq. (4), it follows that $D_i \geq \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N = D_i^v$.

Stable critical state. During the stable critical state, the subtasks of task τ_i are executed using any work-conserving scheduling algorithm on μ_i^O dedicated processors. The total work and the critical-path length of any job of task τ_i during the critical state is at most C_i^O and L_i^O, respectively. Based on Lemma 2, the maximum time job J_i takes to finish its execution starting from its release at time 0 is $L_i^O + (C_i^O - L_i^O)/\mu_i^O$. We will show that $L_i^O + (C_i^O - L_i^O)/\mu_i^O \leq D_i$, which implies that job J_i meets its deadline during the stable critical state. Since $\omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)$, from Eq. (6) we have

$$D_i \geq \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^O} + L_i^O$$

$$\Leftrightarrow \quad D_i \geq \frac{C_i^O - L_i^O}{\mu_i^O} + (C_i^N - L_i^N) \cdot (\frac{1}{\mu_i^N} - \frac{1}{\mu_i^O}) + L_i^O$$

(Since $C_i^N \geq L_i^N$ because total work includes the work on the critical path and $\mu_i^O \geq \mu_i^N$, we have $(C_i^N - L_i^N) \cdot (1/\mu_i^N - 1/\mu_i^O) \geq 0$)

$$\Rightarrow \quad D_i \geq \frac{C_i^O - L_i^O}{\mu_i^O} + L_i^O$$

State Switching. For this case, the job J_i does not complete execution by its virtual deadline D_i^v and it switches from typical to critical state at time D_i^v. The subtasks of job J_i execute on μ_i^N processors during the interval $[0, D_i^v)$ and on μ_i^O processors after time D_i^v.

Let ℓ be the cumulative length of intervals in $[0, D_i^v)$ during which at least one of the μ_i^N dedicated processors assigned to job J_i of task τ_i is idle such that $0 \leq \ell \leq D_i^v$. Since J_i is not finished by time D_i^v and because at least one processor is idle for a duration of ℓ time units in $[0, D_i^v)$, the length of the critical path by time D_i^v is decreased by at least ℓ time units. The remaining length of the critical path at time D_i^v, denoted by L_{remain}, is at most

$$L_{remain} = (L_i^O - \ell) \tag{7}$$

where L_i^O is the overload critical-path length of τ_i. The cumulative length of intervals in $[0, D_i^v)$ during which all the μ_i^N processors are simultaneously busy is $(D_i^v - \ell)$. Therefore, the amount of work done before the task τ_i switches its state at time D_i^v is at least $[\ell + \mu_i^N \cdot (D_i^v - \ell)]$. The remaining amount of total work at time D_i^v, denoted by C_{remain}, is at most

$$C_{remain} = C_i^O - [\ell + \mu_i^N \cdot (D_i^v - \ell)] = C_i^O - \ell - \mu_i^N \cdot (D_i^v - \ell) \tag{8}$$

where C_i^O is the overload total work of task τ_i. Since the total remaining work includes the remaining work of the critical path, we have

$$
\begin{aligned}
& L_{remain} \leq C_{remain} \\
& \big(\text{From Eq. (7) and Eq. (8)}\big) \\
\Leftrightarrow\ & L_i^O - \ell \leq C_i^O - \ell - \mu_i^N \cdot (D_i^v - \ell) \\
& \Big(\text{Since } D_i^v = \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N \text{ from Eq. (4)}\Big) \\
\Leftrightarrow\ & L_i^O - \ell \leq C_i^O - \ell - \mu_i^N \cdot (L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N} - \ell) \\
\Leftrightarrow\ & \mu_i^N \cdot L_i^N + L_i^O - C_i^O + C_i^N - L_i^N \leq \mu_i^N \cdot \ell \\
\Leftrightarrow\ & L_i^N - \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N} \leq \ell \\
& (\text{Since } 0 \leq \ell) \\
\Rightarrow\ & max\left\{0, L_i^N - \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N}\right\} \leq \ell \\
\Leftrightarrow\ & L_i^N - max\left\{0, L_i^N - \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N}\right\} \geq L_i^N - \ell \\
\Leftrightarrow\ & min\left\{L_i^N, \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^N}\right\} \geq L_i^N - \ell \\
& \big(\text{Since } \omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)\big) \\
\Leftrightarrow\ & min\left\{L_i^N, \frac{\omega_i}{\mu_i^N}\right\} \geq L_i^N - \ell
\end{aligned} \tag{9}
$$

Since μ_i^O processors are assigned to job J_i from time D_i^v, the job J_i completes its execution no later than time $D_i^v + L_{remain} + \frac{C_{remain} - L_{remain}}{\mu_i^O}$ according to Lemma 2. We will show that $D_i^v + L_{remain} + \frac{C_{remain} - L_{remain}}{\mu_i^O} \leq D_i$, which implies that J_i completes at or before its deadline. We have to prove that the following holds:

$$
\begin{aligned}
& D_i^v + L_{remain} + \frac{C_{remain} - L_{remain}}{\mu_i^O} \leq D_i \\
& (\text{From Eq. (7) and Eq. (8)}) \\
\Leftrightarrow\ & D_i^v + L_i^O - \ell + \frac{[C_i^O - \ell - \mu_i^N \cdot (D_i^v - \ell)] - (L_i^O - \ell)}{\mu_i^O} \leq D_i \\
\Leftrightarrow\ & D_i^v + L_i^O - \ell + \frac{C_i^O - \mu_i^N \cdot (D_i^v - \ell) - L_i^O}{\mu_i^O} \leq D_i \\
& (\text{Since } D_i^v = \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^N \text{ from Eq. (4)} \\
\Leftrightarrow\ & L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^O - \ell + \frac{C_i^O - \mu_i^N \cdot \left(L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N}\right) + \mu_i^N \cdot \ell - L_i^O}{\mu_i^O} \leq D_i \\
\Leftrightarrow\ & L_i^N + \frac{C_i^N - L_i^N}{\mu_i^N} + L_i^O - \ell + \frac{(C_i^O - C_i^N) - (L_i^O - L_i^N)}{\mu_i^O} - \frac{\mu_i^N \cdot (L_i^N - \ell)}{\mu_i^O} \leq D_i \\
& (\text{Since } \omega_i = (C_i^O - C_i^N) - (L_i^O - L_i^N)) \\
\Leftrightarrow\ & \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O + (L_i^N - \ell) \cdot (1 - \frac{\mu_i^N}{\mu_i^O}) \leq D_i
\end{aligned}
$$

$$\text{(From Eq. (9), } min\left\{L_i^N, \frac{\omega_i}{\mu_i^N}\right\} \geq (L_i^N - \ell))$$
$$\Leftarrow \quad \frac{C_i^N - L_i^N}{\mu_i^N} + \frac{\omega_i}{\mu_i^O} + L_i^O + min\left\{L_i^N, \frac{\omega_i}{\mu_i^N}\right\} \cdot (1 - \frac{\mu_i^N}{\mu_i^O}) \leq D_i$$
$$\Leftrightarrow \quad Eq.\ (5)$$

Therefore, the generic job J_i of HH task τ_i meets its deadline in all the three scenarios. ◀

For each HH task τ_i, we can apply the schedulability test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ in Eq. (5) to determine whether the HH task τ_i meets its deadline in all correct states if μ_i^N and μ_i^O number of dedicated processors are assigned during the typical and critical states, respectively. We say that $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$=TRUE if Eq. (5) is satisfied; otherwise $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$=FALSE. The salient feature of the schedulability test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ is that the set of all possible pairs (μ_i^N, μ_i^O) where $1 \leq \mu_i^N \leq \mu_i^O \leq M$ for which HH task τ_i is deemed schedulable in all the correct states can be determined. The elements in each such pair are potential final values of π_i^N and π_i^O for task τ_i. To this end, we define $\overline{\Omega}(\tau_i)$ the set of all such valid pairs (μ_i^N, μ_i^O) for which the HH task τ_i is schedulable in any state as follows:

$$\overline{\Omega}(\tau_i) = \{(\mu_i^N, \mu_i^O) \mid \texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O) = \texttt{TRUE};\ \mu_i^N = 1, 2 \ldots M;\ \mu_i^O = \mu_i^N \ldots M\} \tag{10}$$

We now filter some of the unnecessary elements from set $\overline{\Omega}(\tau_i)$ to limit the number of valid pairs. Consider that $\texttt{SCHH}(\tau_i, 1, 1)$=FALSE, $\texttt{SCHH}(\tau_i, 1, 2)$=TRUE and $\texttt{SCHH}(\tau_i, 1, 3)$=TRUE for some HH task τ_i. Based on Eq. (10), we have $(1, 1) \notin \overline{\Omega}(\tau_i)$ and $\{(1, 2), (1, 3)\} \subseteq \overline{\Omega}(\tau_i)$. However, we may discard the element $(1, 3)$ from set $\overline{\Omega}(\tau_i)$ since when μ_i^N=1 it is unnecessary (wastage of resource) to consider μ_i^O=3 because μ_i^O=2 processors are enough to guarantee the correctness of task τ_i during the critical state. Therefore, we only need to consider such pair $(\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i)$ where $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1)$=FALSE. To this end, we define $\Omega(\tau_i)$ the set of pairs (μ_i^N, μ_i^O) from set $\overline{\Omega}(\tau_i)$ for which $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1)$=FALSE as follows:

$$\Omega(\tau_i) = \{(\mu_i^N, \mu_i^O) \mid (\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i); \texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1) = \texttt{FALSE}\} \tag{11}$$

Note that Eq. (5) can be tested in constant time for the given values of C_i^N, L_i^N, C_i^O, L_i^O, μ_i^N and μ_i^O for HH task τ_i. The set $\overline{\Omega}(\tau_i)$ in Eq. (10) can be computed for task τ_i by applying test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O)$ at most $M(M+1)/2$ times since $1 \leq \mu_i^N \leq \mu_i^O \leq M$. Therefore, the time complexity to compute the set $\overline{\Omega}(\tau_i)$ for one HH task τ_i is $O(M^2)$. Since $1 \leq \mu_i^N \leq \mu_i^O \leq M$, the number of elements in $\overline{\Omega}(\tau_i)$ is $O(M^2)$. For all the $O(M^2)$ elements in set $\overline{\Omega}(\tau_i)$, we can test $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1) = \texttt{FALSE}$ is Eq. (11) is time $O(M^2)$. Therefore, set $\Omega(\tau_i)$ can be computed in time $O(M^2)$. Since for each element $(\mu_i^N, \mu_i^O) \in \Omega(\tau_i)$ we have $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O - 1)$=FALSE, it follows that the number of elements in set $\Omega(\tau_i)$ is $O(M)$ and the set $\Omega(\tau_i)$ can be computed in time $O(M^2)$.

▶ **Lemma 6.** *If $(\mu_i^N, \mu_i^O) \in \Omega(\tau_i)$, then the* HH *task τ_i meets all its deadlines if the number of dedicated processors during the typical and critical states are μ_i^N and μ_i^O, respectively.*

Proof. Since $(\mu_i^N, \mu_i^O) \in \Omega(\tau_i)$ only if $(\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i)$ based on Eq. (11). Moreover, if $(\mu_i^N, \mu_i^O) \in \overline{\Omega}(\tau_i)$, then $\texttt{SCHH}(\tau_i, \mu_i^N, \mu_i^O) = \texttt{TRUE}$ based on Eq. (10). Therefore, task τ_i meets all its deadlines based on Lemma 5 if the number of dedicated processors during the typical and critical states are μ_i^N and μ_i^O, respectively. ◀

▶ **Example 7.** Consider the two HH tasks in Table 1 and M=8. We only list few elements of set $\Omega(\tau_i)$ in Table 1 for simplicity of presentation. The values in Table 1 will be used later.

Table 1 Example of two HH tasks and some elements (μ_i^N,μ_i^O) in $\Omega(\tau_i)$ computed using Eq. (11)

Task	C_i^N	L_i^N	C_i^O	L_i^O	D_i	Some elements in $\Omega(\tau_i)$
τ_1	9	4	52	20	45	{(1,2), (2,2), (3,3)}
τ_2	11	4	80	42	54	{(2,6),(3,4)}

Given the set $\Omega(\tau_i)$ for each task $\tau_i \in \Gamma_{\mathtt{HH}}$, we now determine the total number of processors required for correctly scheduling *all* the HH tasks in each state. Our objective is to find different alternatives to assign ***all*** the HH tasks to the processors using the different alternatives in $\Omega(\tau_i)$ for each HH task τ_i.

Without loss of generality assume that there are Q number of HH tasks in set Γ such that $Q = |\Gamma_{\mathtt{HH}}|$ and the indices of the HH tasks in set $\Gamma_{\mathtt{HH}}$ ranges from 1 to Q such that $\Gamma_{\mathtt{HH}} = \{\tau_1, \tau_2, \ldots \tau_Q\}$. We also define sequence $\mathcal{S}_{\mathtt{HH}}^p =< \tau_1, \tau_2, \ldots \tau_p >$ that includes the HH tasks with indices from 1 to p, for $p = 1, 2, \ldots Q$. Note that the sequence $\mathcal{S}_{\mathtt{HH}}^Q$ includes all the tasks in $\Gamma_{\mathtt{HH}}$. Given the sequence of p tasks in $\mathcal{S}_{\mathtt{HH}}^p$, we denote $\xi(\mathcal{S}_{\mathtt{HH}}^p)$ as the set where

- each element in set $\xi(\mathcal{S}_{\mathtt{HH}}^p)$ is a *pair of sequences* such that for each such pair of sequences
  - each sequence has p numbers;
  - the i^{th} element in the first sequence is the number of processors required to meet the deadline of the i^{th} HH task in sequence $\mathcal{S}_{\mathtt{HH}}^p$ during the typical state, and
  - the i^{th} element in the second sequence is the number of processors required to meet the deadline of the i^{th} HH task in sequence $\mathcal{S}_{\mathtt{HH}}^p$ during the critical state.

For example, consider $\xi(\mathcal{S}_{\mathtt{HH}}^3)$ ={(<1,2,3>,<4,5,6>), (<2,2,4>, <3,5,5>)} for the three tasks in sequence $\mathcal{S}_{\mathtt{HH}}^3 =< \tau_1, \tau_2, \tau_3 >$. The interpretation of set $\xi(\mathcal{S}_{\mathtt{HH}}^3)$ ={(<1,2,3>,<4,5,6>), (<2,2,4>, <3,5,5>)} is the following:

- There are two elements in set $\xi(\mathcal{S}_{\mathtt{HH}}^3)$. Each of the two elements $(< 1, 2, 3 >, < 4, 5, 6 >)$ and $(< 2, 2, 4 >, < 3, 5, 4 >)$ is a pair of sequences, where each sequence in a pair has $p = 3$ numbers.
- The pair $(< 1, 2, 3 >, < 4, 5, 6 >)$ specifies that the number of dedicated processors required for task τ_1 (which is the 1^{st} task in sequence $\mathcal{S}_{\mathtt{HH}}^3$) during the typical and critical states are 1 and 4, respectively. Similarly, the number of dedicated processors required for task τ_3 in sequence $\mathcal{S}_{\mathtt{HH}}^3$ for the typical and critical states are 3 and 6, respectively. The total number of processors for all the three tasks in set $\mathcal{S}_{\mathtt{HH}}^3$ for the typical and critical state are $(1 + 2 + 3) = 6$ and $(4 + 5 + 6) = 15$, respectively.

Each element in set $\mathcal{S}_{\mathtt{HH}}^Q$ specifies a particular alternative to assign *all* the HH tasks from sequence $\mathcal{S}_{\mathtt{HH}}^Q$ to the processors so that the deadlines for all the HH tasks during the typical and critical states are met. After the set $\xi(\mathcal{S}_{\mathtt{HH}}^Q)$ is computed, we select one alternative from set $\xi(\mathcal{S}_{\mathtt{HH}}^Q)$ so that the capacity constraint for *all* the tasks in Γ is satisfied. Next we present how to compute set $\xi(\mathcal{S}_{\mathtt{HH}}^Q)$.

4.2.1 Computing $\xi(\mathcal{S}_{\mathtt{HH}}^Q)$

We apply dynamic programming to find set $\xi(\mathcal{S}_{\mathtt{HH}}^Q)$. The sum of the p numbers in the first sequence and the sum of the p numbers in the second sequence for any element in $\xi(\mathcal{S}_{\mathtt{HH}}^p)$ are respectively the total number of processors required during the typical and critical states for the HH tasks in $\mathcal{S}_{\mathtt{HH}}^p$. Since the number of processors of the platform is M, the total number of processors required for any state must not be larger than M in order to satisfy the capacity constraint. Based on this observation, the set $\xi(\mathcal{S}_{\mathtt{HH}}^Q)$ is recursively computed by considering one-by-one $\texttt{HH}$ task from the sequence $\mathcal{S}_{\texttt{HH}}^{Q}$. In other words, we first compute $\xi(\mathcal{S}_{\texttt{HH}}^{1})$, then we compute $\xi(\mathcal{S}_{\texttt{HH}}^{2})$, and continuing in this fashion, we finally compute $\xi(\mathcal{S}_{\texttt{HH}}^{Q})$.

The set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ for $p = 1$ is computed as follows:

$$\xi(\mathcal{S}_{\texttt{HH}}^{1}) = \xi(< \tau_1 >) = \{(< a >, < b >) \mid (a, b) \in \Omega(\tau_1)\ \} \tag{12}$$

where $\Omega(\tau_1)$ is given in Eq (11). By assuming that the set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ is already computed, the set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ is recursively computed for $p = 2, 3, \ldots Q$ as follows:

$$\xi(\mathcal{S}_{\texttt{HH}}^{p}) = \Big\{ \big(< a_1, \ldots a_{p-1}, a_p >, < b_1, \ldots b_{p-1}, b_p > \big) \mid \texttt{COND1} \wedge \texttt{COND2} \wedge \texttt{COND3} \wedge \texttt{COND4} \Big\} \tag{13}$$

where

$\texttt{COND1}$: $(< a_1, \ldots a_{p-1} >, < b_1, \ldots b_{p-1} >) \in \xi(\mathcal{S}_{\texttt{HH}}^{p-1})$

$\texttt{COND2}$: $(a_p, b_p) \in \Omega(\tau_p)$

$\texttt{COND3}$: $(a_1 + \ldots + a_{p-1} + a_p) \leq M$ and $(b_1 + \ldots + b_{p-1} + b_p) \leq M$

$\texttt{COND4}$: If $(a_1 + \ldots + a_p) \neq (c_1 + \ldots + c_p)$ for some $(< c_1, \ldots c_p >, < d_1, \ldots d_p >) \in \xi(\mathcal{S}_{\texttt{HH}}^{p})$, then add $(< a_1, \ldots a_p >, < b_1, \ldots b_p >)$ in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$; otherwise, if $(a_1 + \ldots + a_p) = (c_1 + \ldots + c_p)$ and $(b_1 + \ldots + b_p) < (d_1 + \ldots + d_p)$ for some $(< c_1, \ldots c_p >, < d_1, \ldots d_p >) \in \xi(\mathcal{S}_{\texttt{HH}}^{p})$, then add $(< a_1, \ldots a_{p-1}, a_p >, < b_1, \ldots b_{p-1}, b_p >)$ in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ and remove $(< c_1, \ldots c_p >, < d_1, \ldots d_p >)$ from set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$.

Discussion. The set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ in Eq. (13) is computed by selecting each element $(< a_1, \ldots, a_{p-1} >, < b_1, \ldots b_{p-1} >)$ from $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ due to $\texttt{COND1}$ and each element (a_p, b_p) from $\Omega(\tau_p)$ due to $\texttt{COND2}$ such that $(a_1 + \ldots + a_{p-1} + a_p) \leq M$ and $(b_1 + \ldots + b_{p-1} + b_p) \leq M$ due to $\texttt{COND3}$. A new element $(< a_1, \ldots a_{p-1}, a_p >, < b_1, \ldots b_{p-1}, b_p >)$ is added to set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ only if $\texttt{COND4}$ is true, i.e., there is no other element $(< c_1, \ldots c_{p-1}, c_p >, < d_1, \ldots d_{p-1}, d_p >)$ that is already in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ such that $(a_1 + \ldots + a_{p-1} + a_p) = (c_1 + \ldots + c_{p-1} + c_p)$ and $(b_1 + \ldots + b_{p-1} + b_p) \geq (d_1 + \ldots + d_{p-1} + d_p)$.

The $\texttt{COND4}$ ensures that for any two elements $(< a_1, \ldots a_{p-1} >, < b_1, \ldots b_{p-1} >)$ and $(< c_1, \ldots c_{p-1}, c_p >, < d_1, \ldots d_{p-1}, d_p >)$ where $(a_1 + \ldots + a_{p-1} + a_p) = (c_1 + \ldots + c_{p-1} + c_p)$, the element with smaller total number of processors for the critical state is included in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ while the other element is not included in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ (i.e., removed if included previously). Consequently, for a given total number of processors required for the tasks in sequence $\mathcal{S}_{\texttt{HH}}^{p}$ for the typical state, there is at most one element in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$. Since $\texttt{COND3}$ is satisfied, there are at most M different possibilities for the total number of processors required for the tasks for the typical state. Therefore, the number of elements in set $\xi(\mathcal{S}_{\texttt{HH}}^{p})$ is at most $O(M)$. We have the following Lemma 8.

▶ **Lemma 8.** *If $(< a_1, a_2, \ldots a_Q >, < b_1, b_2, \ldots b_Q >) \in \xi(\mathcal{S}_{HH}^{Q})$, then the p^{th} HH task τ_p in sequence $\mathcal{S}_{HH}^{Q}$ meets the deadline in typical and critical states if a_p and b_p dedicated processors are assigned to τ_p respectively during the typical and critical states for $p = 1, 2, \ldots Q$.*

Proof. If $(< a_1, a_2, \ldots a_Q >, < b_1, b_2, \ldots b_Q >) \in \xi(\mathcal{S}_{\texttt{HH}}^{Q})$, then the pair $(a_p, b_p) \in \Omega(\tau_p)$ due to $\texttt{COND2}$ for $p = 1, 2, \ldots Q$. Based on Lemma 6, it holds for each $(a_p, b_p) \in \Omega(\tau_p)$ that task τ_p meets its deadline during the typical and critical state if a_p and b_p dedicated processors are allocated to τ_p during the typical and critical state, respectively. ◀

Time Complexity to find $\xi(\mathcal{S}_{\texttt{HH}}^{Q})$. The time complexity to compute $\xi(\mathcal{S}_{\texttt{HH}}^{Q})$ is $O(n \cdot (n + M) \cdot M^2)$. Recall that there are $O(M)$ elements in $\Omega(\tau_i)$ for each τ_i in $\mathcal{S}_{\texttt{HH}}^{Q}$ (discussed after Eq. (11)). Therefore, the base in Eq. (12) can be computed for task τ_1 in time $O(M)$ since each element $(a_1, b_1) \in \Omega(\tau_i)$ is stored in set $\xi(\mathcal{S}_{\texttt{HH}}^{1})$=$\xi(< \tau_1 >)$ as $(< a >, < b >)$.

The COND4 guarantees that there are at most $O(M)$ elements in set $\xi(\mathcal{S}_{\texttt{HH}}^k)$ for $k = 1, \dots Q$. During each step of the recursion the set $\xi(\mathcal{S}_{\texttt{HH}}^p)$ is computed by considering one element from $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and one element from $\Omega(\tau_p)$. Since there are at most $O(M)$ elements in each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$, the time-complexity to select all the possible ways to select one element from each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$ (i.e., applying COND1 and COND2) is $O(M^2)$. And, there are $O(M^2)$ possible choices to select one element from each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$.

For each of these $O(M^2)$ selections, we apply COND3 and COND4. Given a selection $(< a_1, \dots a_{p-1} >, < b_1, \dots b_{p-1} >) \in \xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $(a_p, b_p) \in \Omega(\tau_p)$, we can apply COND3 in time $O(n)$ since there are $2(p-1) = O(n)$ additions to evaluate COND3. We then apply COND4 in time $O(M)$ since there can be at most $O(M)$ elements already included in $\xi(\mathcal{S}_{\texttt{HH}}^p)$ and the sums in COND4 are already computed during this step of the recursion. Consequently, for all the $O(M^2)$ ways to select one element from each set $\xi(\mathcal{S}_{\texttt{HH}}^{p-1})$ and $\Omega(\tau_p)$, the set $\xi(\mathcal{S}_{\texttt{HH}}^p)$ is computed in time $O((n+M) \cdot M^2)$ during the p^{th} recursive step. Since there are at most Q=O(n) tasks in sequence $\mathcal{S}_{\texttt{HH}}^Q$, the set $\xi(\mathcal{S}_{\texttt{HH}}^Q)$ can be computed in time $O(n \cdot (n+M) \cdot M^2)$.

▶ **Example 9.** Consider two HH tasks in Table 1 and M=8 where $\Omega(\tau_1)$ ={(1,2), (2,2), (3,3)} and $\Omega(\tau_2)$ ={(2,6), (3,4)} for $\mathcal{S}_{\texttt{HH}}^2 =< \tau_1, \tau_2 >$.

Based on Eq (12), we have $\xi(< \tau_1 >)$={(<1>,<2>), (<2>,<2>), (<3>,<3>)}. We will now show how to find $\xi(< \tau_1, \tau_2 >)$ based on Eq. (13). There are total $3 \times 2 = 6$ ways to select one element from each set $\xi(< \tau_1 >)$ and $\Omega(\tau_2)$ by applying COND1 and COND2. Therefore, set $\xi(< \tau_1, \tau_2 >)$ *without* applying COND3 and COND4 is

$$\begin{aligned}\xi(< \tau_1, \tau_2 >) = \{&(< 1, 2 >, < 2, 6 >), (< 1, 3 >, < 2, 4 >), (< 2, 2 >, < 2, 6 >),\\ &(< 2, 3 >, < 2, 4 >), (< 3, 2 >, < 3, 6 >), (< 3, 3 >, < 3, 4 >)\}\end{aligned}$$

After applying COND3, the element $(< 3, 2 >, < 3, 6 >)$ is not included in set $\xi(< \tau_1, \tau_2 >)$ since $(3 + 6) > M = 8$. After applying COND4, the element $(< 2, 2 >, < 2, 6 >)$ is not included in set $\xi(< \tau_1, \tau_2 >)$ since there is another element $(< 1, 3 >, < 2, 4 >)$ such that $(2 + 2) = (1 + 3)$ and $(2 + 6) > (2 + 4)$. Therefore, we have

$$\begin{aligned}\xi(< \tau_1, \tau_2 >) = \{&(< 1, 2 >, < 2, 6 >), (< 1, 3 >, < 2, 4 >),\\ &(< 2, 3 >, < 2, 4 >), (< 3, 3 >, < 3, 4 >)\}\end{aligned}$$

4.3 Overall Task Assignment: Capacity Constraint

In this subsection, we determine whether there is an assignment of all the tasks to the processors such that the total number of processors required during each of the two states is not larger than M. We will now determine a set, denoted by Π, which is a subset of $\xi(\mathcal{S}_{\texttt{HH}}^Q)$ using which it can be verified whether the capacity constraint at each state for all the tasks is satisfied or not. The set Π is defined as follows:

$$\Pi = \begin{cases} \emptyset & \text{if } Q > 0 \text{ and } \xi(\mathcal{S}_{\texttt{HH}}^Q) = \emptyset \\ \{(< a_1, \dots a_Q >, < b_1, \dots b_Q >) \mid \texttt{COND5} \land \texttt{COND6}\} & \text{otherwise} \end{cases} \quad (14)$$

where

COND5: $(< a_1, \dots a_Q >, < b_1, \dots b_Q >) \in \xi(\mathcal{S}_{\texttt{HH}}^Q)$

COND6: $(a_1 + \dots + a_Q + \sum_{\tau_i \in \Gamma_{\texttt{LH}}} \pi_i^N + \Pi_{\texttt{LU}}) \le M$ and $(b_1 + \dots + b_Q + \Pi_{\texttt{LU}}) \le M$.

▶ **Theorem 10.** *The MCFQ scheduling algorithm correctly schedules all the tasks in set Γ if $\Pi \neq \emptyset$.*

Proof. Since $\Pi \neq \emptyset$, we have at least one pair $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \Pi$ such that `COND5` and `COND6` are satisfied. Since $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \xi(\Gamma_{\mathtt{HH}})$ according to `COND5`, each `HH` task τ_i meets its deadline in both typical and critical state if it is assigned a_i and b_i processors according to Lemma 8.

Each `LH` task τ_i requires π_i^N dedicated processors to ensure its correctness according to Eq. (3) of Lemma 3. Therefore, the total number of dedicated processors for all the `LH` tasks to ensure their correctness is $\sum_{\tau_i \in \Gamma_{\mathtt{LH}}} \pi_i^N$. The total number of processors required for scheduling all the low-utilization tasks during typical and critical state is $\Pi_{\mathtt{LU}}$, where $\Pi_{\mathtt{LU}}$ is the minimum number of processors required by the MC-Partition-0.75 to schedule all the low-utilization tasks in set $(\Gamma_{\mathtt{HL}} \cup \Gamma_{\mathtt{LL}})$.

Therefore, the total number of processors for all the tasks is $(a_1 + \ldots + a_Q + \sum_{\tau_i \in \Gamma_{\mathtt{LH}}} \pi_i^N + \Pi_{\mathtt{LU}})$ and $(b_1 + \ldots + b_Q + \Pi_{\mathtt{LU}})$ respectively for the typical and critical state. Since $(a_1 + \ldots + a_Q + \sum_{\tau_i \in \Gamma_{\mathtt{LH}}} \pi_i^N + \Pi_{\mathtt{LU}}) \leq M$ and $(b_1 + \ldots + b_Q + \Pi_{\mathtt{LU}}) \leq M$ based on `COND6`, the capacity constraint at each state is met, the task assignment declares success, and the system correctly schedules all the tasks based on `MCFQ` algorithm. ◀

4.4 Improving the QoS of LH Tasks

The set Π in Eq. (14) provides different alternatives to assign all the tasks to the processors by assuming that all the `LH` tasks are dropped during critical state. However, if there are unused processors during the critical state, then such unused processors may be allocated to the `HH` tasks rather than dropping the `LH` tasks during critical state. Based on this observation, we propose a scheme to maximize the number of `LH` tasks that are never dropped.

We select the alternative from set Π that minimizes the total number of processors required during the critical state for all the `HH` tasks. Let $(< a_1, \ldots a_Q >, < b_1, \ldots b_Q >) \in \Pi$ is the alternative that minimizes the total number of processors required during the critical state for all the `HH` tasks. The number of unused processors during the critical state, denoted by Π_{idle}, is computed as $\Pi_{idle} = M - (\sum_{i=1}^{Q} b_i + \Pi_{\mathtt{LU}})$.

The unused processors can be allocated to the `HH` task τ_i when it does not complete by its virtual deadline and requires additional $(\pi_i^O - \pi_i^N)$ processors to ensure its correctness. By allocating the unused processors to the `HH` tasks, we may not need to drop some or any of the `LH` tasks. Given that there are Π_{idle} unused processors during the critical state, we formulate an ILP to maximum the number of `LH` tasks that are never dropped.

Let $x_i \in \{0, 1\}$ denote a decision variable whether the `LH` task τ_i may need to be dropped or not. If $x_i = 1$, then the `LH` task τ_i is never dropped and will be assigned $\pi_i^O = \pi_i^N$ dedicated processors also during the critical state. If $x_i = 0$, then the `LH` task τ_i may need to be dropped and we set $\pi_i^O = 0$. The value of decision variable x_i for $\tau_i \in \Gamma_{\mathtt{LH}}$ is determined using the following ILP to maximum the number of `LH` tasks that are never dropped:

$$\begin{aligned} &\underset{x_i}{\text{maximize}} && \sum_{\tau_i \in \Gamma_{\mathtt{LH}}} x_i \\ &\text{subject to} && \sum_{\tau_i \in \Gamma_{\mathtt{LH}}} \pi_i^N \cdot x_i \leq \Pi_{idle} \quad \textbf{and} \quad (x_i = 0 \textbf{ or } x_i = 1) \end{aligned} \tag{15}$$

Given the values of x_i for all the `LH` tasks, the fraction of the total number of `LH` tasks that are never dropped is $(\sum_{\tau_i \in \Gamma_{\mathtt{LH}}} x_i)/|\Gamma_{\mathtt{LH}}|$ and is the measure of the QoS for a given taskset under `MCFQ` algorithm. We can also improve the QoS of the `LL` tasks by allocating them to such idle processors based on partitioned EDF scheduling for sequential tasks (not addressed in this paper).

5 Empirical Investigation

The recent work by Li et al. [26] proposed the `MCFS-Improve` schedulability test for federated scheduling of `MC` parallel tasks. In this section, we present the effectiveness of our proposed schedulability test in Theorem 10 (denoted by `Our-MCFQ`) in guaranteeing the schedulability and improving the QoS of randomly generated `MC` parallel tasks in comparison to the state-of-the-art `MCFS-Improve` test in [26]. Before we present our results, we present the taskset generation algorithm.

5.1 Taskset Generation Algorithm

Since both `Our-MCFQ` and `MCFS-Improve` tests depend only on the total work and the critical-path length of each parallel task, we will directly generate these two parameters for each parallel task. We denote U^N and U^O respectively the total nominal utilization of all the tasks and total overload utilization of all the `HI`-critical tasks in a randomly generated taskset Γ such that $U^N = \sum_{\tau_i \in \Gamma} u_i^N$ and $U^O = \sum_{\tau_i \in (\Gamma_{\texttt{HH}} \cup \Gamma_{\texttt{HL}})} u_i^O$. Let $U_B = max\{U^N/M, U^O/M\}$ denotes the upper bound on normalized total system utilization. Note that $U_B \leq 1$ is a necessary condition for schedulability of taskset Γ on M processors.

The following experimental parameters are used for generating a random `MC` sporadic DAG taskset with normalized total system utilization U_B for M processors:

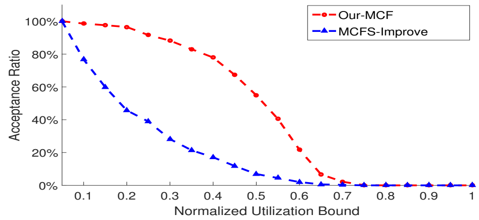
- The proportion of high-utilization tasks in a taskset is controlled using probability p^{hu}.
- The overload utilization of each high-utilization task is controlled using u_{max}.
- The ratio of the period and overload critical-path length of task τ_i is controlled using a parameter P_{max} such that $1 \leq T_i/L_i^O \leq P_{max}$.
- The proportion of `HI`-critical tasks is controlled using probability p^{hc}.
- The ratio of overload and nominal utilizations of task τ_i is controlled using a parameter R_{max} such that $1 \leq u_i^O/u_i^N \leq R_{max}$.

The following values of the experimental parameters are used:

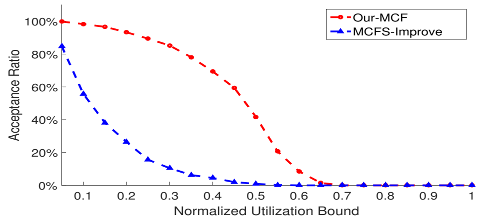
- Number of processors: $M \in \{16, 32, 48, 64, 80, 96, 112, 128, 144, 160\}$.
- Normalized utilization bound: $U_B \in \{0.05, 0.1, \ldots 1.0\}$.
- Probability of a task to be a high-utilization task: $p^{hu} \in \{0.1, 0.2, \ldots 1.0\}$.
- Upper bound on overload utilization of a high-utilization task: $u_{max} \in \{2.0, 4.0, \ldots 16.0\}$.
- The maximum ratio of period and overload critical-path length: $P_{max} \in \{2.0, 2.25 \ldots 4.0\}$.
- Probability of a task to be a `HI`-critical task: $p^{hc} \in \{0.1, 0.2, \ldots 1.0\}$.
- The maximum ratio of overload and nominal utilizations: $R_{max} \in \{2.0, 2.25 \ldots 4.0\}$.

We consider a total of 12,960,000 different combinations of the above parameters to generate the tasksets. For each combination, we generate 1000 parallel `MC` tasksets where each taskset is generated as follows (each parameter is selected from an uniform distribution):

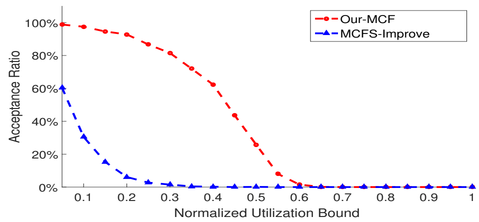
- Task period $D_i = T_i$ is drawn from the range $[10, 1000]$.
- A real number p_i^u is drawn from the range $[0, 1]$. If $p_i^u \leq p^{hu}$, then τ_i is a high-utilization task and its overload utilization u_i^O is drawn in the range $[1.02, u_{max}]$; otherwise, τ_i is a low-utilization task and its overload utilization u_i^O is drawn in the range $[0.02, 1]$. The overload total work of τ_i is $C_i^O = u_i^O \times T_i$.
- A real number P_i is drawn from the range $[1, P_{max}]$ and the overload critical-path length is $L_i^O = T_i/P_i$.
- A real number p_i^c is drawn from the range $[0, 1]$. If $p_i^c \leq p^{hc}$, then $Z_i = $ `HI`; otherwise $Z_i = $ `LO`.
- If $Z_i = $ `HI`, then a real number R_i is drawn from the range $[1, R_{max}]$; otherwise $R_i = 1$.



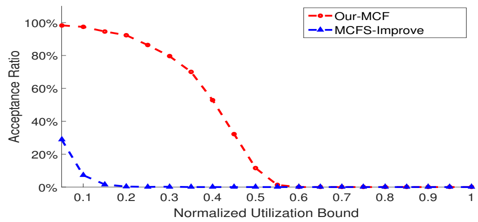
(a) $M = 16$



(b) $M = 32$



(c) $M = 64$



(d) $M = 128$

Figure 1 Comparison of acceptance ratios for different number of processors for $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$.

- The nominal total work and critical-path length are $C_i^N = C_i^O/R_i$ and $L_i^N = L_i^O/R_i$, respectively.
- Repeat the above steps as long as $max\{U^O/m, U^N/m\} \leq U_B$. Once the condition is violated, discard the task that was generated the last.
- If the resulting taskset satisfies the condition $max\{U^O/m, U^N/m\} > U_B - 0.05$, then accept the taskset and stop the procedure. Otherwise, discard the taskset and the repeat the above steps.

The above taskset generation procedure ensures that each taskset has a total normalized utilization within the range $U_B - 0.05$ and U_B. This is reasonable because in our experiments we consider values of U_B that are incremented in step of 0.05.

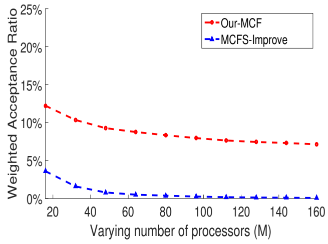
5.2 Results: Schedulability Tests

We compare the effectiveness of `Our-MCFQ` test in terms of guaranteeing the schedulability of randomly generated parallel `MC` tasksets in comparison to the `MCFS-Improve` test in [26].

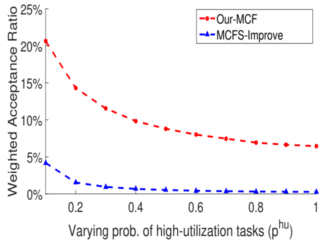
For a given schedulability test and values of M, U_B, p^{hu}, u_{max}, P_{max}, p^{hc} and R_{max}, let the *acceptance ratio* denotes the fraction of tasksets out of 1000 tasksets that are deemed schedulable by the test at normalized utilization bound U_B. The acceptance ratios for $M = 16, 32, 64, 128$ are presented in Figure 1 for $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$ where the x-axis is the normalized utilization bound U_B and the y-axis is the acceptance ratio.

The acceptance ratios of both tests decreases as the normalized utilization bound U_B increases. Such decreasing trend in acceptance ratio for larger U_B is expected because tasksets with a relatively larger utilization are generally difficult to schedule.

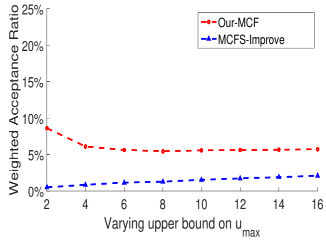
The acceptance ratio of `Our-MCFQ` test is significantly better than the acceptance ratio of `MCFS-Improve` test for $M = 16, 32, 64, 128$. For example, the acceptance ratio in Figure 1b at $U_B = 0.4$ for $M = 32$ is around 70% for `Our-MCFQ` test and less than 10% for `MCFS-Improve` test. For $M = 128$ in Figure 1d, the acceptance ratio at $U_B = 0.2$ is



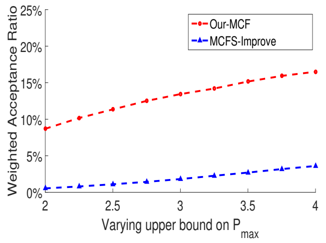
(a) Varying M



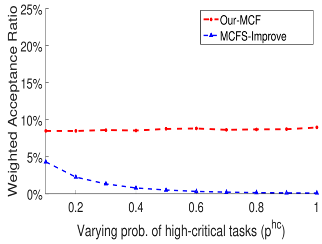
(b) varying p^{hu}



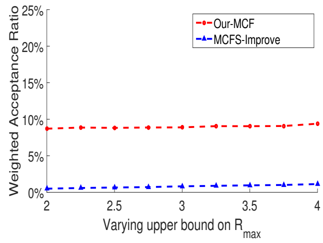
(c) varying u_{max}



(d) Varying P_{max}



(e) Varying p^{hc}



(f) Varying R_{max}

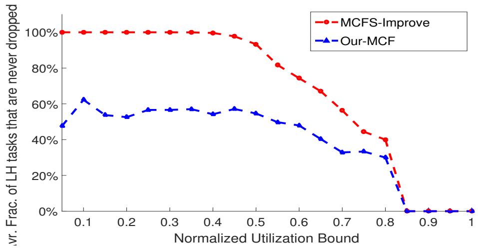
Figure 2 Weighted acceptance ratios for varying values of M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}.

around 90% for `Our-MCFQ` test and 0% for `MCFS-Improve` test. The acceptance ratio of the `MCFS-Improve` test decreases to zero very rapidly with increasing U_B for higher number of processors in comparison to the `Our-MCFQ` test.

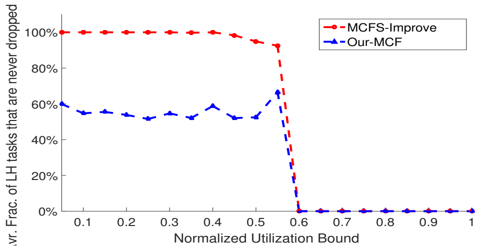
The relatively higher acceptance ratio of the `Our-MCFQ` test is due to our proposed task assignment algorithm for the `HH` tasks. The `MCFQ` algorithm determines an assignment of the `HH` tasks to the processors by choosing from different alternatives by taking in to account the number of processors required for other tasks during the typical and critical states. On the other hand, the task assignment of the `MCFS-Improve` test is restrictive in terms of the number of different alternatives for assigning the `HH` tasks to the processors. It can be analytically shown that if we plugin the alternative for assigning processors to the `HH` tasks computed based on the `MCFS-Improve` test into the proposed schedulability test in Eq. (5), then the test in Eq. (5) is also satisfied, which implies that the capacity augmentation bound of the `MCFS-Improve` test also applies to our proposed test. However, such an analysis is omitted in this paper due to space constraint.

The results presented in [26] show quite high acceptance ratio in comparison to the results presented in this paper for the `MCFS-Improve` test. The reason is that we do not use the task set generation algorithm from [26] because some of the assumptions were not explicitly described in [26]. For example, it is not described in [26] how random numbers with log normal distribution with mean $(1 + \sqrt{m}/3)$ was generated without knowing the mean (μ) and standard deviation (σ) of the associated normal distribution.

For comparison of the acceptance ratios of `Our-MCFQ` test and `MCFS-Improve` test for varying values of M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}, we also computed the *weighted acceptance ratios* and presented in Figure 2. The weighted acceptance ratio denotes the fraction of schedulable tasksets weighted by the normalized utilization bound U_B. If $AR(U_B)$ denotes the acceptance ratio of a schedulability test for normalized utilization bound U_B for some given values of M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}, then the weighted acceptance ratio for a set S of U_B values is given as follows: $W(S) = \left(\sum_{U_B \in S}(AR(U_B) \times U_B)\right) / \sum_{U_B \in S} U_B$.



(a) $M = 16$



(b) $M = 32$

Figure 3 Average fraction of `LH` tasks that are never dropped for $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$.

When computing the weighted acceptance ratio by varying one parameter, the other five parameters are kept fixed. The fixed values of the parameters are $M = 64$, $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$ and $R_{max} = 2.0$. The significantly higher weighted acceptance ratio of `Our-MCFQ` test in comparison to `MCFS-Improve` test is evident in Figure 2a-2f respectively for the variation of the parameters M, p^{hu}, u_{max}, P_{max}, p^{hc}, and R_{max}. The acceptance ratio of `Our-MCFQ` is much higher because the task assignment algorithm is successful in finding an allocation of the tasks to the processors such that the system is correct while the task assignment of the `MCFS-Improve` test fails in many cases to find such an assignment.

5.3 Results: Quality of Service

In this subsection, we compare the effectiveness of `Our-MCFQ` test with `MCFS-Improve` test in improving the QoS of the system in terms of average fraction of the number of `LH` tasks that are not dropped regardless of the state of the system. Note that `Our-MCFQ` test can significantly schedule more tasksets than the `MCFS-Improve` test (Figure 1). For fairness, we compare the QoS for only those tasksets that are deemed schedulable by both tests.

For each taskset that is deemed schedulable using both the `Our-MCFQ` test and the `MCFS-Improve` test, (i) we apply the ILP in Eq. (15) to determine the fraction of the number of `LH` tasks that are never dropped under the `MCFQ` algorithm, and (ii) we also determine the fraction of the number of `LH` tasks that are never dropped based on the implementation in [26]. The average fraction of the number of `LH` tasks that are never dropped (over all the tasksets that are schedulable by both test) at each normalized utilization bound U_B is computed for each test and presented for $M = 32$ and $M = 64$ in Figure 3 where $p^{hu} = 0.5$, $u_{max} = 2.0$, $P_{max} = 2.0$, $p^{hc} = 0.5$, and $R_{max} = 2.0$.

It is evident that `Our-MCFQ` test is able to schedule all the `LH` tasks for normalized utilization $U_B \leq 0.4$ while `MCFS-Improve` is never successful in allocating all the `LH` tasks for any U_B. For $U_B > 0.4$, the `Our-MCFQ` test can also schedule large fraction of the `LH` tasks without ever dropping them in comparison to `MCFS-Improve`. Therefore, the QoS of the system using `Our-MCFQ` test is much higher than that of under `MCFS-Improve`.

6 Related Work

There have been several works on real-time scheduling of parallel non-MC tasks on multiprocessors based on fork-join model [22, 1], synchronous parallel task model [35, 32, 15], and the dag task model [10, 12, 28, 3, 31]. Many of these works proposed resource-augmentation bounds and schedulability tests for global scheduling where the nodes of the tasks are allowed to migrate from one processor to another. There are two other mechanisms to schedule parallel DAG tasks: federated scheduling [25] and decomposition-based scheduling [21]. In decomposition-based scheduling, a DAG task is transferred into a set of independent sporadic task by inserting artificial release time and artificial deadline. The decomposed subtasks of all the DAG tasks are scheduled based on GEDF scheduling policy in [21].

There are many works on scheduling MC systems since the seminal work by Vestal who first proposed the MC sequential task model and its analysis based on fixed-priority scheduling algorithm on uniprocessor platform [38]. Building upon Vestal's seminal work [38], there have been several approaches [9, 16, 11, 8, 24, 19, 4, 17, 23, 7, 5, 34] to design certification-cognizant scheduling of MC system for both uni- and multiprocessor. The work in [14] presents a recent survey on real-time scheduling of MC sequential tasks. To improve the quality of service for the LO-critical tasks, there are also works that consider that the LO-critical tasks are not dropped but provide delayed results, for example, by executing them less frequently after the system switches to the critical state (e.g., weakly hard MC task model [18], elastic MC task model [37, 36, 20]) or provides imprecise results [29, 13, 5, 34].

There are very few works on scheduling MC parallel tasks. Some works considers time-table based scheduling [2] or partitioned MC scheduling based on decomposition strategy [30]. However, such scheduling algorithms are not applicable to DAG tasks for which the internal structure is only known during runtime. The work in [27] and its extension in [26] consider federated scheduling of MC sporadic DAG tasks. The authors in [27, 26] also derived capacity augmentation bound of 3.67 for dual-critical tasks. It is also shown that the schedulability test based on the capacity augmentation bound in [27, 26] does not perform well in comparison to the schedulability test MCFS-Improve that is based on actual assignment of the tasks to the processors. However, the task assignment for each HH task in MCFS-Improve algorithm is not aware of how the other tasks are assigned to the processors and may fail to assign all the tasks to the processors even if there is another way to successfully assign the tasks. On the other hand, our proposed MCFQ algorithm does not finalize the assignment when analyzing each HH task rather finalize the assignment when analyzing the overall task assignment for all the tasks.

7 Conclusion

This paper presents a new schedulability analysis for federated scheduling of MC sporadic DAG tasks on multiprocessors. The salient feature of this analysis is that different alternatives to allocate each of the HH tasks to the processors during the typical and critical states of the system are considered. The particular alternative to allocate a HH task is selected such that all the tasks can be correctly scheduled on a given number of processors. The MCFQ algorithm also tries to maximize the fraction of the number of LH tasks that are never dropped. Experimental results show that the proposed schedulability test for MCFQ algorithm not only can schedule much larger number of random tasksets but also can improve the QoS of the system significantly in comparison to the state of the art. Investigating the schedulability of MC parallel tasks where more than one high-utilization tasks are scheduled on a set of dedicated processors is an interesting future work.

References

1 P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of ECRTS*, 2013.

2 S. Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *Proc. of RTNS*, 2012.

3 S. Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *Proc. of ECRTS*, 2014.

4 S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *Proc. of ECRTS*, 2012. doi:10.1109/ECRTS.2012.42.

5 S. Baruah, A. Burns, and Z. Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *Proc. of ECRTS*, 2016. doi:10.1109/ECRTS.2016.12.

6 S. Baruah, B. Chattopadhyay, H. Li, , and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Syst.*, 50(1):142–177, 2014. doi:10.1007/s11241-013-9184-2.

7 S. Baruah, A. Eswaran, and Z. Guo. MC-Fluid: Simplified and Optimally Quantified. In *Proc. of RTSS*, 2015. doi:10.1109/RTSS.2015.38.

8 S. Baruah, Haohan Li, and L. Stougie. Towards the Design of Certifiable Mixed-criticality Systems. In *Proc. of RTAS*, 2010. doi:10.1109/RTAS.2010.10.

9 S. Baruah and S. Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proc. of ECRTS*, pages 147–155, 2008.

10 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proc. of RTSS*, 2012.

11 Sanjoy Baruah, Alan Burns, and Robert Davis. Response-time analysis for mixed criticality systems. In *Proc. of RTSS*, 2011.

12 V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *Proc. of ECRTS*, 2013.

13 A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Proc. of WMC, RTSS*, 2013.

14 A. Burns and R. Davis. Mixed-criticality systems: A review. In *(available online), Tenth Edition*, January, 2018. URL: http://www-users.cs.york.ac.uk/~burns/review.pdf.

15 Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, A. Easwaran, and Insik Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *Proc. of ECRTS*, 2013.

16 François Dorin, Pascal Richard, Michaël Richard, and Joël Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46:305–331, 2010.

17 P. Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proc. of the ECRTS*, 2012.

18 Oliver Gettings, Sophie Quinton, and Robert I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proc. of RTNS*, 2015. doi:10.1145/2834848.2834850.

19 Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. of RTSS*, 2011. doi:10.1109/RTSS.2011.10.

20 Mathieu Jan, Lilia Zaourar, and Maurice Pitel. Maximizing the execution rate of low-criticality tasks in mixed criticality systems. In *Proc. of WMC, RTSS*, 2013. URL: http://www-users.cs.york.ac.uk/~robdavis/wmc2013/paper6.pdf.

21 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.

22 K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. of RTSS*, 2010.

23 J. Lee, K. M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee. MC-Fluid: Fluid Model-Based Mixed-Criticality Scheduling on Multiprocessors. In *Proc. of RTSS*, 2014. doi:10.1109/RTSS.2014.32.

24 Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proc. of RTSS*, pages 183–192, 2010.

25 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.

26 J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5):760–811, Sep 2017. doi:10.1007/s11241-017-9281-8.

27 J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu. Mixed-criticality federated scheduling for parallel real-time tasks. In *Proc. of RTAS*, April 2016. doi:10.1109/RTAS.2016.7461340.

28 Jing Li, K. Agrawal, Chenyang Lu, and C. Gill. Analysis of global edf for parallel tasks. In *Proc. of ECRTS*, 2013.

29 Di Liu, Jelena Spasic, Gang Chen, Nan Guan, Songran Liu, Todor Stefanov, and Wang Yi. EDF-VD Scheduling of Mixed-Criticality Systems with Degraded Quality Guarantees. In *Proc. of RTSS*, 2016. doi:10.1109/RTSS.2016.013.

30 Guangdong Liu, Ying Lu, Shige Wang, and Zonghua Gu. Partitioned multiprocessor scheduling of mixed-criticality parallel jobs. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2014. doi:10.1109/RTCSA.2014.6910497.

31 A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G.C. Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *Proc. of ECRTS*, 2015.

32 G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proc. of ECRTS*, pages 321–330, July 2012. doi:10.1109/ECRTS.2012.37.

33 OpenMP. Openmp application program interface. version 4.0. 2013.

34 R. Pathan. Improving the quality-of-service for scheduling mixed-criticality systems on multiprocessors. In *Proc. of ECRTS*, 2017.

35 A. Saifullah, K. Agrawal, Chenyang Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proc. of RTSS*, 2011.

36 H. Su, N. Guan, and D. Zhu. Service guarantee exploration for mixed-criticality systems. In *Proc. of RTCSA*, 2014. doi:10.1109/RTCSA.2014.6910499.

37 H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proc. of DATE*, 2013. doi:10.7873/DATE.2013.043.

38 S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proc. of RTSS*, pages 239–243, 2007. doi:10.1109/RTSS.2007.47.