Noname manuscript No.
(will be inserted by the editor)

Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks

Jing Li · David Ferry · Shaurya Ahuja · Kunal Agrawal · Christopher Gill · Chenyang Lu

Received: date / Accepted: date

Abstract A mixed-criticality system comprises safety-critical and non-safety-critical tasks sharing a computational platform. Thus, different levels of assurance are required by different tasks in terms of real-time performance. As the computational demands of real-time tasks increase, tasks may require internal parallelism in order to complete within stringent deadlines. In this paper, we consider the problem of mixed-criticality scheduling of parallel real-time tasks and propose a novel mixed-criticality federated scheduling (MCFS) algorithm for parallel tasks modeled by a directed acyclic graph. MCFS is based on federated intuition for scheduling parallel real-time tasks. It strategically assigns cores and virtual deadlines to tasks to achieve good schedulability.

For high-utilization tasks (utilization ≥ 1), we prove that MCFS provides a *capacity augmentation bound* of $2+\sqrt{2}$ and $(5+\sqrt{5})/2$ for dual- and multi-criticality, respectively. We show that MCFS has a capacity augmentation bound of $11m/(3m-3)$ for dual-criticality systems with both high- and low-utilization tasks. For high-utilization tasks, we further provide a MCFS-Improve algorithm that has the same bound but can admit many more task sets in practice. Results of numerical experiments show that MCFS-Improve significantly improves over MCFS for many different workload settings.

We also present an implementation of a MCFS runtime system in Linux that supports parallel programs written in OpenMP. Our implementation provides graceful degradation and recovery features. We conduct empirical experiments to demonstrate the practicality of our MCFS approach.

Keywords Mixed-Criticality Federated Scheduling · Capacity Augmentation Bound · Mixed-Criticality Real-Time System · Parallel Real-Time Tasks

This research was supported by NSF grants CCF-1136073, CCF-1337218, and CNS-1329861.

Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, Chenyang Lu
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box. 1045, St Louis, MO 63130, USA
E-mail: {li.jing, davidferry, shaurya.ahuja, kunal, cdgill, lu}@wustl.edu

1 Introduction

In ***mixed-criticality*** systems, tasks with different criticality levels share a computing platform and demand different levels of assurance in terms of real-time performance. For example, when an autonomous vehicle is in danger of an accident, crash-avoidance systems are more safety-critical than route planning or stability enhancing systems. On the other hand, in normal driving conditions all these features are essential and need to meet their deadlines to provide a smooth and stable drive, while infotainment systems only need to make the best effort.

The mixed-criticality model is an emerging paradigm for real-time systems, as it can significantly improve resource efficiency. In particular, safety-critical tasks must be approved by some *certification authority (CA)* and their schedulability must be guaranteed under possibly pessimistic assumptions about task execution parameters, while system designers usually try to meet the deadlines of all tasks but with less stringent validation. Both the CA and system designers must make estimates of each task's worst-case execution time (work). Thus, a task τ_i may be characterized by two different worst-case work values: a pessimistic *overload work* C_i^O for certification and a less pessimistic *nominal work* C_i^N from empirical measurements. The goal of mixed-criticality scheduling is two-fold: (1) In the *typical-state* — when all tasks exhibit nominal behavior — all tasks must be schedulable. (2) In the *critical-state* — when some task exceeds its nominal work — all safety-critical tasks must still be schedulable, but we need not guarantee schedulability of other tasks.

Mixed-criticality systems have been extensively studied (see (Burns and Davis, 2016) for a survey). However, most prior research considers sequential tasks, where each task individually can only be executed on a single core at a time, and utilizes multiprocessors by exploiting inter-task parallelism within task sets. Hence, the execution time of any single task is confined by the capacity of a single core. In contrast, parallel tasks are tasks with internal parallelism and can potentially use multiple cores. Tasks with internal parallelism are already used in real-time systems such as autonomous vehicles (Kim et al, 2013) and real-time hybrid structural simulation (Ferry et al, 2014), to support tasks with higher execution requirements or tighter deadlines atop shared multi-core computing platforms.

In this paper, we study the problem of mixed-criticality scheduling of parallel real-time tasks. Three related trends make it increasingly important to accelerate the convergence of mixed-criticality systems and parallel tasks: (1) rapid increases in the number of cores per chip; (2) increasing demand for consolidation and integration of functionality with different levels of criticality on shared multi-core platforms; and (3) increasing computational demands of individual tasks, which makes parallel execution essential to meet deadlines. Although there has been extensive research on the two related problems, namely, mixed-criticality scheduling of sequential tasks (see (Burns and Davis, 2016) for a survey); and single-criticality scheduling of parallel tasks (Andersson and de Niz, 2012; Baruah et al, 2012b; Bonifaci et al, 2013; Chwa et al, 2013; Kim

et al, 2013; Lakshmanan et al, 2010; Li et al, 2013, 2014; Liu and Anderson, 2012; Nelissen et al, 2012; Saifullah et al, 2013). To our knowledge, there has been almost no prior work on the combined problem of mixed-criticality scheduling of parallel tasks, except for (Baruah, 2012b; Liu et al, 2014).

Mixed-criticality parallel tasks: A parallel task can be modeled as a *directed acyclic graph (DAG)*, where nodes represent subtasks and edges between nodes represent precedence constraints. As mentioned above, the system designer and the CA both provide estimates of work — the worst-case execution time on a single core. C_i^N and C_i^O denote the nominal and overload work, respectively. For parallel tasks, they must also provide estimates for critical-path length — the amount of time it takes to complete the task on a hypothetically infinite number of cores. L_i^N and L_i^O denote the nominal and overload critical-path length, respectively.

Capacity augmentation bound for mixed-criticality parallel task sets: Since it is generally impossible to prove utilization bounds for parallel tasks, due to Dhall's effect (Dhall and Liu, 1978), we generalize the idea of *capacity augmentation bounds* (Li et al, 2013) from single- to multi-criticality, to characterize the performance of our scheduler. Informally, a scheduler that can schedule all task sets with the following properties on m cores has a capacity augmentation bound of b: (1) per-state utilization is at most m/b in each state; and (2) each task's critical-path length is at most $1/b$ of its implicit deadline. Note that a capacity augmentation bound implies a resource augmentation bound, but the converse is not true. A capacity augmentation bound generalizes utilization bounds in that it indicates how much over-provisioning is necessary to guarantee schedulability.

Challenges of scheduling mixed-criticality parallel task sets: Incorporating parallelism presents novel challenges for scheduling. The DAG structure of a task may not be known in advance; in fact, each job of the same task may have a different DAG structure. Thus the schedulability analysis and scheduler cannot rely on tasks' structural information. Moreover, tasks may have utilization much smaller than 1 in the typical-state, but much larger than 1 in the critical-state. To deal with this dramatic change, the scheduler must be able to detect the overload behavior early and increase the resource allotment so that the task can still meet its deadline.

Contributions: In this paper, we propose a *mixed-criticality federated scheduling (MCFS)* algorithm and prove its capacity augmentation bound under various conditions. MCFS generalizes federated scheduling (Li et al, 2014) to mixed-criticality systems. Federated scheduling (and thus MCFS) has the advantage of not requiring task decomposition. Thus, the scheduler does not need to know the internal structure of the tasks, a priori. The work and critical-path length estimates of a task give an abstraction of the DAG. Moreover, they can be empirically measured without knowing the specific structures of all the task's instances. We assume that all tasks are *implicit-deadline sporadic tasks*.

This paper is a extended version of an RTAS 2016 paper (Li et al, 2016). In this paper, we expand on the contributions in (Li et al, 2016) with a lower bound of MCFS for dual-criticalty systems, an improved capacity augmentation bound of MCFS for multi-criticality systems, a new feature of the MCFS runtime system, and additional empirical evaluations. In addition, we present an improved MCFS algorithm that has the same capacity augmentation bound for high-utilization tasks but can admit significantly more task sets than MCFS in practice, as shown in the extensive numerical simulations on a wide rage of parameters. In particular, we made the following contributions.

Section 4 provides details of the MCFS algorithm and the basic schedulability test for dual-criticality systems where all tasks are *high-utilization tasks* — that is, all tasks have either nominal or overload utilization larger than 1. We first consider only high-utilization tasks, since parallelism is essential for them to meet their deadlines. For this dual-criticality system with high-utilization tasks, we prove the correctness of MCFS and also prove that it has a capacity augmentation bound of $2 + \sqrt{2} \approx 3.41$ in Section 5. We show that this bound is tight for the MCFS algorithm by providing an example task set requiring an capacity augmentation bound of $\beta = 2 - \frac{3}{2m} + \sqrt{(2 - \frac{3}{2m})^2 - 2}$ for $m > 9$. Note that when $m \to \infty$, $\beta \to 2 + \sqrt{2}$; when $m \to 9$, $\beta \to 3$.

In Section 6, we generalize the MCFS algorithm and schedulability test for high-utilization task systems with more than two criticality levels. We prove that the capacity augmentation bound in this case is $(5 + \sqrt{5})/2 \approx 3.62$. Section 8 considers dual-criticality systems having both high- and low-utilization tasks. In this case, MCFS has a capacity augmentation bound of $\frac{11m}{3(m-1)}$ (≈ 3.67 for large m). To our knowledge, this is the first known augmentation bound for parallel mixed-criticality tasks.

In addition, in Section 7 we provide an improved schedulability test, namely MCFS-Improve, for dual- and multi-criticality high-utilization tasks. MCFS-Improve generally admits task sets with even higher utilizations than that of MCFS. To evaluate the schedulability of MCFS and MCFS-Improve, we conduct experiments for dual-criticality task sets with a wide range of parameters in Section 10. In all the experimented settings, MCFS and MCFS-Improve can admit many more task sets than is indicated by the capacity augmentation bound. Moreover, results show that MCFS-Improve has relative improvements over MCFS from 23% to 223%, in terms of the fraction of schedulable task sets. MCFS-Improve outperforms MCFS when tasks' critical-path lengths are relatively long, the number of cores is large, and the total utilization is high.

We demonstrate the applicability of MCFS by implementing a MCFS runtime system in Linux that supports OpenMP parallel programs (Section 9). In addition to delivering MCFS service to parallel tasks, this system also supports two additional features: (1) graceful degradation — that is, if a high-criticality task enters its overload state, the system need not immediately discard all low-criticality tasks. Instead, it gradually discards low-criticality tasks as needed. (2) recovery from critical- to typical-state — that is, after the job with overload behavior of a high-criticality task completes, the future jobs of this task

can start from the typical-state, allowing the suspended low-criticality tasks (and hence the system) to recover from critical-state. We conduct empirical evaluations to show the practicality of MCFS system (Section 10).

2 System Model and Background

2.1 Task Model

Now we formally define the mixed-criticality parallel real-time task model. Each job (instance of a parallel task) can be modeled as a dynamically unfolding *directed acyclic graph (DAG)*, in which each node represents a sequence of instructions and each edge represents a precedence constraint between nodes. A node is *ready* to be executed when all its predecessors have been executed. Note that each job of a task could be a different DAG — it may be completely different structurally. For each job J_i of τ_i, we consider two parameters: (1) the total work (execution time) $\mathcal{C}_i$ of job J_i is the sum of execution times of all nodes in job J_i's DAG; and (2) the critical-path length $\mathcal{L}_i$ of job J_i is the length of the longest path weighted by node execution times.

We consider a task set τ of n independent sporadic mixed-criticality parallel tasks $\{\tau_1, \tau_2, ..., \tau_n\}$. The tuple $(Z_i, C_i^N, C_i^O, L_i^N, L_i^O, D_i)$ characterizes a task τ_i. Z_i represents the criticality of a task. For example, in dual-criticality systems $Z_i \in \{\text{LO}, \text{HI}\}$, where *HI* (high-criticality) and *LO* (low-criticality) are the two criticality levels. C_i^N and C_i^O are the ***nominal work*** and ***overload work***, respectively. C_i^N is the less pessimistic estimate generally expected to occur during normal operation, while C_i^O is the potentially much more pessimistic estimate considered during the certification process. Similarly, L_i^N and L_i^O are, respectively, the nominal and overload critical-path length estimates. In this paper, we focus on *implicit-deadline sporadic tasks*, where the relative deadline D_i is equal to the minimum inter-arrival time between two consecutive jobs of the same task. For a task τ_i, its nominal utilization is denoted as $u_i^N = C_i^N/D_i$ and its overload utilization is $u_i^O = C_i^O/D_i$.

When a job J_i of task τ_i is released, we do not know its actual work $\mathcal{C}_i$ nor its actual critical-path length $\mathcal{L}_i$ in advance — these are only revealed as J_i executes. If a job J_i has $\mathcal{C}_i \leq C_i^N$ and $\mathcal{L}_i \leq L_i^N$, then we say that the job exhibits *nominal behavior*; otherwise, it exhibits *overload behavior*. We assume that $\mathcal{C}_i$ and $\mathcal{L}_i$ never exceed C_i^O and L_i^O, respectively.

2.2 System Model for Dual-Criticality System

A dual-criticality system (we extend it to multi-criticality system in Section 6) has two types of tasks: low- and high-criticality tasks. Thus this system has two corresponding states: ***typical-state*** (low-criticality mode) and ***critical-state*** (high-criticality mode). Crucially, at runtime the scheduler does not know if a task will exhibit nominal or overload behavior for a particular run. Thus, the

system begins in the typical-state, assuming each job will satisfy its nominal C_i^N and L_i^N. If, however, any job overruns, then the system *transitions* to the critical-state. Jobs of the low-criticality tasks may be discarded, but the scheduler must ensure that all high-criticality jobs can still meet all their deadlines even if they all execute for their overload parameters C_i^O and L_i^O.

For the purpose of determining task set feasibility, the total utilization of a task set in the typical-state is the sum of the nominal utilizations of all tasks: $U^N = \sum_{\tau_i \in \tau} u_i^N$. Similarly, the total utilization of a task set in the critical-state is the sum of the overload utilizations of high-criticality tasks $U^O = \sum_{\tau_i \in \tau \text{ and } Z_i = \text{HI}} u_i^O$.

2.3 Schedulability Conditions for Dual-Criticality Systems

A mixed-criticality scheduler has two components. Given a task set, the *schedulability test* must first decide whether or not to admit this task set on a machine with m cores. If the test admits the task set, then the *runtime scheduler* must guarantee the following mixed-criticality correctness conditions.

Definition 1 A dual-criticality scheduler is correct if for every task set it admits, it satisfies the following two conditions:

(1) If the system stays in the typical-state during its entire execution, all tasks must meet their deadlines.
(2) After the system transitions into critical-state, all high-criticality tasks must meet their deadlines.

The runtime scheduler must transition properly from typical- to critical-state. Since the DAG of a particular job J_i unfolds dynamically during its execution, the runtime scheduler does not know, a priori, whether a particular job will exhibit nominal or overload behavior. Therefore, the runtime scheduler must infer the transition time (if any) of the system from typical- to critical-state dynamically at runtime, based on the execution of the current jobs. Note that low-criticality tasks need not define C_i^O, since the runtime scheduler is allowed to discard all low-criticality tasks in the critical-state. Even if defined, it is not used in our analysis and so we ignore it.

2.4 Dual-Criticality Capacity Augmentation Bound

In (Li et al, 2013) a capacity augmentation bound is defined for parallel tasks with single-criticality. We generalize this definition to dual-criticality parallel tasks with implicit deadlines.

Definition 2 A scheduler provides a *capacity augmentation bound* of b for dual-criticality parallel task systems, if it can schedule any task set which satisfies the following conditions.

(1) The total nominal utilization of all tasks (high and low-criticality) in typical-state $U^N = \sum_{\tau_i \in \tau} C_i^N / D_i \leq m/b$.
(2) The total overload utilization of high-criticality tasks in critical-state $U^O = \sum_{\tau_i \in \tau \text{ and } Z_i = \text{HI}} C_i^O / D_i \leq m/b$.
(3) For all tasks, $L_i^N \leq L_i^O \leq D_i/b$.

Note that no scheduler can guarantee $b < 1$ — thus the capacity augmentation bound, just like utilization bound, provides an indication of how much slack is needed in the system to guarantee schedulability.

3 Background

We now describe a couple of ideas from prior work that MCFS is based on. In particular, MCFS uses important concepts from two different lines of work. It borrows the idea of *virtual deadlines* from work on mixed-criticality scheduling of sequential tasks (Baruah et al, 2012a, 2014) and the idea of *federated scheduling* from single-criticality scheduling of parallel tasks (Li et al, 2014). We survey other related work in Section 12.

Virtual Deadline: MCFS utilizes the idea of *virtual deadline*, first used in single processor mixed-criticality scheduler EDF-VD (Baruah et al, 2012a) and also used in multiprocessor mixed-criticality schedulers (Baruah et al, 2014). At a high level, in these algorithms each high-criticality task is assigned a virtual deadline $D'_i < D_i$. This virtual deadline serves two purposes: (1) It boosts the priority of a high-criticality job so that if the job exhibits overload behaviour, then the scheduler detects it early and transitions into critical-state. (2) It provides enough slack so that after the transition, there is enough time to complete the overload work of the job.

To see how virtual deadlines are used for mixed-critical tasks, we briefly discuss three relevant algorithms, namely, *EDF-VD* (Baruah et al, 2012a) — an algorithm for scheduling dual-criticality tasks on a single processor; and *MC-Global* (referred to as Algorithm Global in (Baruah et al, 2014)) and *MC-Partition* (Baruah et al, 2014) — algorithms for scheduling dual criticality sequential tasks on m identical speed processors. In each of these algorithms, the schedulability test occurs as part of a pre-processing phase prior to run-time. At this time, all tasks are assigned a virtual deadline $D'_i \leq D_i$. Using this virtual deadline assignment, the schedulability of the task set is determined. At run-time, jobs are initially dispatched with the expectation that no job will execute for more than its nominal work and all jobs' deadlines are assumed to be D'_i instead of D_i. During runtime, if some job does execute beyond its nominal work, then the system *transitions* into the critical-state. The run-time scheduling and dispatching algorithms are immediately modified in the following way: (1) All currently-active jobs of low-criticality tasks are immediately discarded; henceforth, no job of a low-criticality task will be allowed to execute. (2) The scheduling algorithm now considers the real deadlines D_i

rather than the virtual deadlines D'_i when making scheduling and dispatching decisions for high-criticality tasks.

The three algorithms differ in the details of their schedulability tests, calculation of the virtual deadlines, and details of their runtime scheduling and dispatching algorithms. EDF-VD is designed for single-processor systems and uses EDF for both low and high-criticality modes of the system. It guarantees a resource augmentation bound of 4/3. MC-Global algorithm extends the scheduling of sequential mixed-criticality workloads from single-processor to the multiprocessor setting and provides a resource augmentation bound of $\sqrt{5}+1$. MC-Partition partitions tasks among processors and then uses EDF-VD on each processor. In particular, a version of MC-Partition, namely MC-Partition-UT-0.75 provides a utilization bound of about 3/8, by incorporating scheduling tasks with utilizations up to 1 — we use this one as a black box in MCFS to schedule tasks whose utilization never exceeds 1.

Federated Scheduling: MCFS is based on the federated scheduling (Li et al, 2014) that assigns dedicated cores to high-utilization tasks and schedules them using a work-conserving scheduler. The key insight of the federated scheduling paradigm is to calculate the minimum number of cores to assign to a job in order to complete its remaining work and meet its deadline, which is given by the following lemma and used throughout this paper.

Lemma 1 *If job J_i needs to execute C'_i remaining work and L'_i remaining critical-path length, it is schedulable by a work-conserving scheduler and can complete its execution within D'_i time on n_i dedicated cores, where $n_i \geq \frac{C'_i - L'_i}{D'_i - L'_i}$.*

Proof This Lemma can be proved using arguments very similar to Theorem 2 in Li et. al (Li et al, 2014). Here, we only provide the intuition. Recall that a work-conserving (greedy) scheduler never keeps a core idle if there is any work available. We say that a time step is *incomplete* if any of the n_i dedicated cores is idle during that time step and the time step is *complete* otherwise. It is straight-forward to see that if the job's remaining critical-path length is at most L'_i, then the total number of incomplete steps is $I \leq L'_i$. In addition, the total number of complete steps is $X \leq \frac{C'_i - L'_i}{n_i} \leq D'_i - L'_i$, if we substitute the value of n_i. Since each step is either complete or incomplete, the total number of time steps to complete the job is $I + X$, which is bounded by D'_i. □

We now describe the basics of the federated scheduling for single-criticality parallel tasks on m identical cores. Each task τ_i has worst case execution time (work) C_i, worst case critical path length L_i, deadline (equal to minimum inter-arrival time) D_i, and utilization $u_i = C_i/D_i$. The federated scheduling algorithm first classifies tasks into either high utilization tasks ($u_i > 1$) or a low-utilization tasks ($u_i \leq 1$). Each high utilization task τ_i is assigned n_i dedicated cores, where n_i is $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. Therefore, the total number of cores assigned to high-utilization tasks is $n_{\text{high}} = \sum_{\tau_i \in \tau_{\text{high}}} n_i$. The remaining $n_{\text{low}} = m - n_{\text{high}}$ cores are assigned collectively to low-utilization tasks. The algorithm admits the task set τ, if n_{low} is non-negative and $n_{\text{low}} \geq 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$.

At runtime, any greedy (work-conserving) parallel scheduler can be used to schedule a high-utilization task τ_i on its assigned n_i cores. Low-utilization tasks are treated as though they are sequential tasks and any multiprocessor scheduling algorithm with a utilization bound of at least 50% can be used to schedule the low-utilization tasks on the allotted n_{low} cores. This approach provides a capacity augmentation bound of 2 — that is, a task set is admitted and meets all deadlines as long as its total utilization is smaller than $m/2$ and for each task τ_i, its worst-case critical-path length $L_i \leq D_i/2$.

4 Scheduling Dual-Criticality High-Utilization Tasks

The MCFS scheduler consists of two parts: (1) a mapping algorithm (including schedulability test) runs before the tasks start executing; and (2) a runtime scheduler executes tasks if the task set is schedulable. Before runtime, the MCFS scheduler tries to generate a mapping for each criticality state: the *typical-state (low-criticality) mapping* S^T and the *critical-state (high-criticality) mapping* S^C. If it cannot find a valid mapping for both the states, then the task set is declared unschedulable. At runtime, the scheduler performs typical-state mapping S^T when all jobs exhibit nominal behavior. If any job exceeds its nominal parameters, then the system transitions into the critical-state and the scheduler switches to using mapping S^C.

In this section, we only consider dual-criticality systems where all tasks are high-utilization tasks. Intuitively, a task is a high-utilization task, if it requires parallel execution to meet its deadline. We consider task systems that contain both high and low-utilization tasks in Section 8. Table 1 shows the notation used throughout this paper.

Definition 3 A task is a ***high-utilization task*** if it is a high-criticality task with overload utilization larger than 1; or it is a low-criticality task with nominal utilization larger than 1.

4.1 Mapping Algorithm

MCFS computes two quantities for each task: (1) a virtual deadline; and (2) the number of cores assigned to the task in both the typical- and the critical-state. At a high level, the deadline assignment and the core assignment are designed to carefully balance the requirements in both states. To generate a mapping, MCFS classifies tasks into three categories.

1. **LO-High (LH)** tasks are low-criticality tasks with high-utilization in nominal behavior, i.e. $u_i^N > 1$. Again, these tasks are discarded in critical-state.
2. **HI-VeryLow-High (HVH)** tasks are high-criticality tasks with very low utilization $u_i^N \leq 1/(b-1)$ in nominal behavior and high utilization $u_i^O > 1$ in overload behavior.

Table 1 Table of Notations

Symbol	Meaning in the paper
C_i^N (C_i^O)	Nominal (overload) work (or execution time) of task τ_i
L_i^N (L_i^O)	Nominal (overload) critical-path length of task τ_i
u_i^N (u_i^O)	Nominal (overload) utilization of task τ_i
D_i	Implicit deadline of sporadic task τ_i
D_i'	Assigned virtual deadline of τ_i for nominal behavior
n_i^N	Number of assigned cores to τ_i for nominal behavior
n_i^O	Number of assigned cores to τ_i for overload behavior
$\tau_{\mathcal{C}}$	Set of tasks in category $\mathcal{C} \in$ {LH, ..., HVH, HMH}
$U_{\mathcal{C}}^N$ ($U_{\mathcal{C}}^O$)	Total nominal (overload) utilization of category $\mathcal{C}$ tasks
$N_{\mathcal{C}}^N$ ($N_{\mathcal{C}}^O$)	Total #cores assigned to $\tau_{\mathcal{C}}$ for nominal (overload) behavior
$S^{\mathcal{S}}$	Mapping in state $\mathcal{S} \in$ {typical, intermediate, ..., critical}

Table 2 High-Utilization Task Classification

Task Type	Criticality	Nominal Utilization	Overload Utilization
HMH	High	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$
HVH	High	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$
LH	Low	$1 < u_i^N$	NA

Table 3 High-Utilization Task Virtual Deadline and Core Assignment

Task Type	b	Virtual Deadline D_i'	Number of assigned cores n_i^N for nominal behavior	Number of assigned cores n_i^O for overload behavior
HMH	$2+\sqrt{2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N - L_i^N}{D_i' - L_i^N}\right\rceil, \lceil u_i^O \rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O}\right\rceil\right\}$
HVH	$2+\sqrt{2}$	$\frac{D_i}{b-1}$	$\lfloor u_i^O \rfloor$	$\left\lceil\frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O}\right\rceil$
LH	2	D_i	$\left\lceil\frac{C_i^N - L_i^N}{D_i - L_i^N}\right\rceil$	NA

3. **HI-Moderate-High (HMH)** tasks are high-criticality tasks with moderate utilization $u_i^N > 1/(b-1)$ in nominal behavior and high-utilization $u_i^O > 1$ in overload behavior.

Table 2 shows the classification criterion and Table 3 shows the virtual deadline assignments and the core assignments for all the categories. As mentioned earlier, we only consider high-utilization tasks; therefore, the above categories are exhaustive.

4.2 Schedulability Conditions of MCFS

The MCFS scheduler declares a task set schedulable, if and only if it is schedulable in both typical- and critical-states. The schedulability of a task set τ can be determined by the following conditions:

- If $L_i^N \geq D_i'$ for any task, or if $L_i^O \geq D_i - D_i'$ for any high-criticality task, then τ is declared unschedulable.
- If there are not enough cores for the typical-state mapping, i.e. $N_{\text{LH}}^N + N_{\text{HVH}}^N + N_{\text{HMH}}^N > m$, then τ is unschedulable.
- If there are not enough cores for the critical-state mapping, i.e. $N_{\text{HVH}}^O + N_{\text{HMH}}^O > m$, then τ is unschedulable.
- If none of above cases occurs, then τ is schedulable.

In the next section, we will show that if the MCFS schedulability test admits a task set, then the runtime scheduler guarantees meeting the correctness conditions from Definition 1. We will also show that MCFS has a capacity augmentation bound of $2+\sqrt{2} \approx 3.41$ for task sets with high-utilization tasks.

4.3 MCFS Runtime Execution

At runtime, the system is assumed to start in the typical-state and the runtime scheduler executes jobs according to the typical-state mapping S^T — that is, each task is scheduled by a work-conserving scheduler on n_i^N dedicated cores.

If a high-criticality job J_i does not complete within its virtual deadline D_i', then the system transitions into the critical-state. All low-criticality jobs can be abandoned and future jobs of low-criticality tasks need not be admitted. The scheduler now executes all jobs according to their critical-state mapping S^C. That is, all high-utilization tasks are now allocated n_i^O dedicated cores and scheduled using a work-conserving scheduler.

Remark: Note that after transitioning to the critical-state, MCFS *needs not* abandon all low-criticality tasks; it can degrade gracefully by abandoning low-criticality tasks *on demand*. If, for instance, a high-criticality job J_i of τ_i exceeds its virtual deadline, it requires $n_i^O - n_i^N$ additional cores. Then, MCFS only needs to suspend enough low-criticality tasks to free up these cores and it can leave the remaining tasks unaffected. In addition, once job J completes, MCFS can recover from critical-state to typical-state simply by giving n_i^N cores to τ_i and re-admitting the low-criticality tasks that were suspended.

5 Proof of Correctness and Capacity Augmentation Bound

We now prove that for high-utilization tasks MCFS guarantees (1) correctness (as described in Definition 1) and; (2) a capacity augmentation bound of $2+\sqrt{2}$ (as described in Definition 2). We first prove properties of the mappings generated by MCFS (from Table 3) for each of the three categories of tasks

(from Table 2). In particular, we will show that each task of a category is schedulable under the generated mapping and the numbers of cores assigned to the task are bounded in both typical- and critical-state. These properties then allow us to prove correctness and the capacity augmentation bound.

Before diving into the proofs, we state two simple mathematical inequalities that will be used throughout the proofs.

(1) If $\frac{a}{b} \geq c > 0$ and $0 \leq x \leq y < b$, then $\frac{a}{b} \leq \frac{a-cx}{b-x} \leq \frac{a-cy}{b-y}$;
(2) If $0 < \frac{a}{b} \leq c$ and $0 \leq x \leq y < b$, then $\frac{a}{b} \geq \frac{a-cx}{b-x} \geq \frac{a-cy}{b-y}$;

5.1 LH tasks under MCFS

Recall that an LH task τ_i is a low-criticality task with high utilization ($u_i^N > 1$) under nominal condition. Since these tasks may be discarded in critical-state, we need only consider their typical-state behavior where they are assigned with $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D_i - L_i^N} \right\rceil$ dedicated cores (see Table 3) and a virtual deadline $D_i' = D_i$. Given these facts, the following lemma is obvious from Lemma 1.

Lemma 2 *LH tasks are schedulable under MCFS.*

We now prove that the number of cores assigned to a LH task is bounded.

Lemma 3 *For any $b \geq 3$, if a LH task τ_i has $D_i \geq bL_i^N$, then the number of cores it is assigned in the typical-state is bounded by $n_i^N \leq (b-1)u_i^N$.*

Proof A LH task has high utilization; therefore, $u_i^N = C_i^N / D_i > 1$.

Since $L_i^N \leq D_i/b$, using Ineq (1), we have

$$n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D_i - L_i^N} \right\rceil < \frac{C_i^N - L_i^N}{D_i - L_i^N} + 1 \leq \frac{C_i^N - D_i/b}{D_i - D_i/b} + 1$$
$$\leq \frac{bu_i^N - 1}{b-1} + 1 = \frac{bu_i^N + b - 2}{b-1} \leq \frac{bu_i^N + (b-2)u_i^N}{b-1} = 2u_i^N$$

Therefore, $n_i^N \leq (b-1)u_i^N$ for any $b \geq 3$. □

5.2 HVH tasks under MCFS

Recall that a HVH task τ_i is a high-criticality task with very low utilization $u_i^N \leq 1/(b-1)$ in the nominal condition and with high utilization $u_i^O \geq 1$ in the overload condition. Table 3 shows that MCFS assigns a HVH task $\lfloor u_i^O \rfloor$ dedicated cores in the typical-state and increases the number of allocated cores to $n_i^O = \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$ in the critical-state. Its virtual deadline is set as $D_i' = D_i/(b-1)$.

Lemma 4 *HVH tasks are schedulable under MCFS.*

Proof In typical-state, the total work of an HVH task τ_i is $C_i^N = D_i u_i^N \leq D_i/(b-1) = D'_i$. Therefore, a single dedicated core is already enough to complete this work. Since $u_i^O \geq 1$, the number of cores assigned to a HVH task $n_i^N = \lfloor u_i^O \rfloor \geq 1$ is sufficient.

Now let us consider the critical-state. There are two cases:

Case 1: The transition occurred before the release of job j_i. Then the job gets $n_i^O = \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil$ cores. Since $u_i^O > 1$, we have

$$n_i^N = \lfloor u_i^O \rfloor \leq u_i^O = \frac{C_i^O}{D_i} \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$$

Then by applying Ineq (1), we get

$$n_i^O = \left\lceil \frac{C_i^O - L_i^O - n_i^N D'_i}{D_i - L_i^O - D'_i} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil$$

Thus, n_i^O are sufficient by Lemma 1.

Case 2: The transition to critical-state occurred during the execution of a job j_i of task τ_i. Say j_i was released at time r_i and the transition to critical-state occurred at time $t \leq r_i + D'_i$ (If t is larger, then j_i would have finished executing already).

Let $e = t - r_i \leq D'_i$ be the duration for which the job executes before the transition. In these e time steps before the transition, say that the job has t^* complete steps (where all cores are busy working) and $e - t^*$ incomplete steps (where the critical-path length decreases). By definition, $t^* \leq e \leq D'_i$. Then, at the transition, it has at most $C^O - n_i^N t^* - e + t^*$ remaining work and $L_i^O - e + t^*$ remaining critical-path length that must be completed in $D_i - e$ time steps. By Lemma 1, τ_i is guaranteed to complete by the deadline, if τ_i is allocated at least n dedicated cores, where:

$$n = \left\lceil \frac{(C_i^O - n_i^N t^* - e + t^*) - (L_i^O - e + t^*)}{(D_i - e) - (L_i^O - e + t^*)} \right\rceil = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil$$

Similar to Case 1, since $n_i^N \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$ and $t^* \leq D'_i$, by Ineq (1) we have

$$n_i^O = \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil \geq \left\lceil \frac{C_i^O - n_i^N t^* - L_i^O}{D_i - t^* - L_i^O} \right\rceil$$

By Lemma 1, the n_i^O cores are enough for it to be schedulable. □

We now bound the number of cores assigned to HVH tasks. Since HVH tasks have high overload utilization, the following lemma is true.

Lemma 5 *For an HVH task τ_i, the number of assigned cores in the typical-state is $n_i^N = \lfloor u_i^O \rfloor \leq u_i^O$.*

Lemma 6 *For an HVH task τ_i, if $D_i \geq bL_i^O \geq bL_i^N$, then the number of cores assigned in the critical-state is $n_i^O \leq bu_i^O$, for all $\frac{7+\sqrt{33}}{4} \leq b \leq \frac{5+\sqrt{17}}{2}$.*

Proof Since HVH task τ_i satisfies $u_i^O > 1$, we can derive that $n_i^N \geq 1$ and $n_i^N = \lfloor u_i^O \rfloor \leq u_i^O \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$. Therefore, by Ineq (1) and $L_i^O \leq D_i/b$ we get

$$\begin{aligned} n_i^O &< \frac{C_i^O - n_i^N \frac{D_i}{b-1} - L_i^O}{D_i - \frac{D_i}{b-1} - L_i^O} + 1 \leq \frac{C_i^O - n_i^N \frac{D_i}{b-1} - \frac{D_i}{b}}{D_i - \frac{D_i}{b-1} - \frac{D_i}{b}} + 1 \\ &= \frac{b(b-1)u_i^O - bn_i^N - (b-1) + b^2 - 3b + 1}{b^2 - 3b + 1} \end{aligned}$$

Note that $\forall b \in [\frac{5-\sqrt{17}}{2}, \frac{5+\sqrt{17}}{2}]$ we have $b^2 - 5b + 2 \leq 0$. In addition, for $b \geq \frac{7+\sqrt{33}}{4} > \frac{3+\sqrt{5}}{2}$ it is true that $b^2 - 3b + 1 > 0$ and $b^2 - 3b + 2 > 0$.

Now let us consider two cases:

Case 1: If $u_i^O < 2$, then $n_i^N = 1$. Hence, by Ineq (1) we can derive

$$\begin{aligned} n_i^O &< \frac{b(b-1)u_i^O + b^2 - 5b + 2}{b^2 - 3b + 1} \\ &< \frac{b(b-1)u_i^O + (b^2 - 5b + 2)\frac{u_i^O}{2}}{b^2 - 3b + 1} \qquad [\text{since } b^2 - 5b + 2 \leq 0 \text{ and } \tfrac{u_i^O}{2} < 1] \\ &= \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u_i^O \end{aligned}$$

Case 2: If $u_i^O \geq 2$, then $n_i^N = \lfloor u_i^O \rfloor > u_i^O - 1$. By Ineq (1), we can derive

$$\begin{aligned} n_i^O &< \frac{b(b-1)u_i^O - b(u_i^O - 1) - (b-1) + b^2 - 3b + 1}{b^2 - 3b + 1} \\ &= \frac{b(b-2)u_i^O + b^2 - 3b + 2}{b^2 - 3b + 1} \\ &\leq \frac{b(b-2)u_i^O + (b^2 - 3b + 2)\frac{u_i^O}{2}}{b^2 - 3b + 1} \qquad [\text{since } b^2 - 3b + 2 > 0 \text{ and } \tfrac{u_i^O}{2} \geq 1] \\ &= \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u_i^O \end{aligned}$$

Therefore, in both cases, we have

$$n_i^O < \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u_i^O \tag{3}$$

By solving $\frac{3b^2-7b+2}{2(b^2-3b+1)} \leq b$, which is equivalent to $2b^2 - 7b + 2 \geq 0$ for $b > 1$, we can conclude that for all $b \geq \frac{7+\sqrt{33}}{4} \approx 3.19$, we have $n_i^O \leq bu_i^O$.

Finally, by intersecting all the ranges of b, we get the required result $n_i^O \leq bu_i^O$, for all $\frac{7+\sqrt{33}}{4} \leq b \leq \frac{5+\sqrt{17}}{2}$. □

Remark: The classification and core assignment to HVH tasks may seem strange at first glance. Since the tasks have such a low utilization in the typical-state, why do we assign dedicated cores rather than assigning multiple tasks to

each core? This is due to balancing core assignments in the typical- and critical-state. Intuitively, if tasks share cores (basically assigning fewer cores per task) in the typical-state, then we must assign more cores in the critical-state. In particular, note that Lemma 6 uses the fact that the task has dedicated cores to prove a lower bound on the amount of work this task completes by its virtual deadline, allowing us to upper bound the amount of left-over work in the critical-state. If we didn't assign dedicated cores, then such a lower bound would be difficult to prove; therefore, MCFS would have to assign more cores to these tasks in the critical-state, giving a worse bound.

5.3 HMH tasks under MCFS

Recall that a HMH task τ_i is a high-criticality task with moderate or high utilization $u_i^N > 1/(b-1)$ in the nominal condition and with high utilization $u_i^O > 1$ in the overload condition. Table 3 shows that MCFS assigns each HMH task $n_i^N = \max\left\{\left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil, \lceil u_i^O \rceil\right\}$ dedicated cores in the typical-state and increases the number of allocated cores to $n_i^O = \max\left\{n_i^N, \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil\right\}$ in the critical-state. Its virtual deadline is $D_i' = 2D_i/b$.

Lemma 7 *HMH tasks are schedulable under MCFS.*

Proof In the typical-state, by Lemma 1 we know that $\left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil$ is sufficient for a HMH task τ_i to complete its C_i^N and L_i^N within its virtual deadline $2D_i/b$. Thus, n_i^N cores are enough for it to be schedulable in the typical-state.

Now let us consider the critical-state. There are two cases:

Case 1: The transition occurred before the release of j_i.

For $n_i^O = \max\left\{n_i^N, \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil\right\}$, there are two sub-cases:

(a) If $n_i^N > \frac{C_i^O - L_i^O}{D_i - L_i^O}$, then by n_i^N being integer and Ineq (2), we get

$$n_i^O = n_i^N \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil$$

(b) If $n_i^N \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$, similarly by applying Ineq (1), we get

$$n_i^O = \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil \geq n_i^N$$

Hence, in both sub-cases, we have $n_i^O \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil$. Thus, by Lemma 1, n_i^O cores is sufficient to complete its work C_i^O and critical-path length L_i^O within deadline D_i.

Case 2: The transition to critical-state occurred during the execution of a job j_i of task τ_i. Say j_i was released at time r_i and the transition to critical-state occurred at time $t \leq r_i + D_i'$ (If t is larger, then j_i would have finished executing already).

Let $e = t - r_i \leq D'_i$ be the duration for which the job executes before the transition. In these e time steps before the transition, say that the job has t^* complete steps (where all cores are busy working) and $e - t^*$ incomplete steps (where the critical-path length decreases). By definition, $t^* \leq D'_i$. Similar to Case 2 in Lemma 4, τ_i is guaranteed to complete by the deadline, if τ_i is allocated at least $n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil$ dedicated cores.

Again, there are two cases:

(a) If $n_i^N > \frac{C_i^O - L_i^O}{D_i - L_i^O}$, then by $t^* \leq D'_i$ and Ineq (2), we get

$$n_i^O = n_i^N \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil \geq n \geq \left\lceil \frac{C_i^O - L_i^O - n_i^N D'_i}{D_i - L_i^O - D'_i} \right\rceil$$

(b) If $n_i^N \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$, then by $t^* \leq D'_i$ and Ineq (1) we get

$$n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil \leq \left\lceil \frac{C_i^O - L_i^O - n_i^N D'_i}{D_i - L_i^O - D'_i} \right\rceil \leq n_i^O$$

Since $n_i^O \geq n$ in both cases, n_i^O cores are enough for a HMH job j_i to be schedulable if the transition happens during its execution. □

In the following two lemmas, we bound the numbers of cores assigned to HMH tasks in both the typical- and critical-state.

Lemma 8 *For each HMH task τ_i, if $D_i \geq bL_i^N$, then the number of cores assigned in typical-state is bounded by $n_i^N \leq (b-1)u_i^N + u_i^O$, for any $b \geq 2$.*

Proof HMH task τ_i satisfies $u_i^N > 1/(b-1)$ and $u_i^O > 1 \geq 2/b$, for $b \geq 2$. We consider three cases for u_i^N and n_i^N:

Case 1. $\mathbf{u_i^N \leq 2/b}$:

Since $C_i^N \leq 2D_i/b$, from Inequality (2), we have

$$\frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \leq \frac{C_i^N}{2D_i/b} \leq 1 \leq u_i^O \Rightarrow \left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil \leq \lceil u_i^O \rceil$$

Since $1/(b-1) < u_i^N$, we know $(b-1)u_i^N > 1$. We can derive

$$n_i^N = \lceil u_i^O \rceil < 1 + u_i^O < (b-1)u_i^N + u_i^O$$

Case 2. $\mathbf{u_i^N > 2/b}$ and $\left\lceil \frac{\mathbf{C_i^N - L_i^N}}{\mathbf{2D_i/b - L_i^N}} \right\rceil \leq \lceil \mathbf{u_i^O} \rceil$:

This case is similar to Case 1; we also have $n_i^N = \lceil u_i^O \rceil < (b-1)u_i^N + u_i^O$.

Case 3. $\mathbf{u_i^N > 2/b}$ and $\left\lceil \frac{\mathbf{C_i^N - L_i^N}}{\mathbf{2D_i/b - L_i^N}} \right\rceil > \lceil \mathbf{u_i^O} \rceil$:

In this case, since $C_i^N \geq 2D_i/b$ and $L_i^N \leq D_i/b$, from Inequality (1),

$$n_i^N = \left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil < \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} + 1 \leq \frac{C_i^N - D_i/b}{2D_i/b - D_i/b} + 1 = bu_i^N$$

Since $u_i^N \leq u_i^O$, we get $n_i^N \leq bu_i^N \leq (b-1)u_i^N + u_i^O$. □

Lemma 9 *For a HMH task τ_i, if $D_i \geq bL_i^O \geq bL_i^N$, then the number of cores assigned in the critical-state is bounded by $n_i^O \leq bu_i^O$, for all $4 > b \geq 2 + \sqrt{2}$.*

Proof We denote $n' = \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$ and we will show that $n' < bu_i^O$.

$$\begin{aligned}
n' &= \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \\
&\leq \left\lceil \frac{C_i^O - u_i^O D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \qquad [\text{since } n_i^N \geq \lceil u_i^O \rceil \geq u_i^O] \\
&= \left\lceil \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} \right\rceil \qquad [\text{since } D_i' = 2D_i/b] \\
&< \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} + 1
\end{aligned}$$

Since HMH task satisfies $u_i^O > 1$, so we have

$$C_i^O - \frac{2C_i^O}{b} = (1 - \frac{2}{b})u_i^O D_i > (1 - \frac{2}{b})D_i = D_i - \frac{2D_i}{b}$$

Again by applying Inequality (1), we can get

$$\begin{aligned}
n' &\leq \frac{C_i^O - 2C_i^O/b - D_i/b}{D_i - 2D_i/b - D_i/b} + 1 \qquad [\text{since } L_i^O \leq D_i/b] \\
&= \frac{(b-2)u_i^O - 1}{b-3} + 1 \\
&< \frac{(b-2)u_i^O}{b-3} \qquad [\text{since } 1 - \tfrac{1}{b-3} < 0,\ \forall 4 > b > 3] \qquad (4) \\
&\leq \frac{b(b-3)u_i^O}{b-3} \qquad [\text{since } b(b-3) \geq (b-2),\ \forall b \geq 2 + \sqrt{2}] \\
&= bu_i^O
\end{aligned}$$

In Lemma 8, we know that $n_i^N \leq (b-1)u_i^N + u_i^O \leq bu_i^O$. Thus, by intersecting all the ranges of b, we get the required result $n_i^O = \max\left\{n_i^N, n'\right\} \leq bu_i^O$, for all $4 > b \geq 2 + \sqrt{2}$. □

Remark: At a high-level, the intuition behind the allocation is similar to HVH tasks, albeit more complex. Clearly, n_i^N must be enough to schedule the nominal work; we need $n_i^N \geq \left\lceil \frac{C_i^N - L_i^N}{D_i' - L_i^N} \right\rceil$. In addition, we must balance the utilization in typical- and critical-states by providing a lower bound on the amount of work done in the typical-state. In the first line of Lemma 9, we calculate this lower bound by using the fact that $n_i^N \geq u_i^O$. The overload assignment is somewhat more straightforward; we must assign enough cores to complete the remaining work after the mode transition, i.e. $n_i^O \geq \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$. In addition, we must be careful that the number of cores *does not decrease* after the transition.

5.4 Proof of Correctness

Theorem 1 *MCFS is correct — if the MCFS schedulability test declares a task set schedulable, then the runtime scheduler guarantees that the conditions in Definition 1 are met for all possible executions of the task system.*

The correctness is obvious from Lemmas 2, 4 and 7.

Remark: Note that the correctness of MCFS does not rely on a particular b. In fact, the schedulability test in Section 4.2 can use any $b > 2$ to check the schedulability. In fact, it can use different b's for different tasks. If the test passes for any set of b's for various tasks, then the task set is schedulable. Therefore, in principle, one can do an exhaustive search using different values of b's for different tasks to check for schedulability. The particular values of b we used in our description are only in order to provide the capacity augmentation bound. Based on this observation, we provide an improved schedulability test and mapping algorithm for these high-utilization tasks in Section 7.

5.5 Proof of Capacity Augmentation Bound $2 + \sqrt{2}$

We now show a capacity augmentation bound for a particular $b = 2 + \sqrt{2}$; that is, if the total utilizations in both the typical and critical-state are no more than m/b and the critical-path lengths in both the nominal and overload condition are no more than the deadline divided by b for all tasks, then MCFS always declares the task set schedulable.

We first define some notations, summarized in Table 1. The total nominal utilization of all tasks of category $\mathcal{C} \in \{\text{LH, HVH, HMH}\}$ is denoted by $U_{\mathcal{C}}^N$ and their total overload utilization is $U_{\mathcal{C}}^O$. Similarly, the total number of cores assigned to tasks in category $\mathcal{C}$ in the typical-state mapping S^T is $N_{\mathcal{C}}^N$ and in the critical-state mapping S^C is $N_{\mathcal{C}}^O$.

Theorem 2 *MCFS has a capacity augmentation bound of $2 + \sqrt{2}$. That is, if the conditions from Definition 2 hold for $b = 2 + \sqrt{2}$, then the task set always satisfies the following conditions (from Section 4.2):*

(1) virtual deadline is valid — any task has $L_i^N \leq D_i'$ and any high-criticality task has $L_i^O \leq D_i - D_i'$;
(2) typical-state mapping is valid — $N_{LH}^N + N_{HVH}^N + N_{HMH}^N \leq m$;
(3) critical-state mapping is valid — $N_{HVH}^O + N_{HMH}^O \leq m$.

We prove the theorem by showing that MCFS satisfies each of the required conditions via the following three lemmas.

Lemma 10 *For any $b > 3$, if $L_i^N \leq L_i^O \leq D_i/b$ (Condition 3 of Definition 2), then the virtual deadline is always valid.*

Proof **LHI tasks:** Virtual deadline $D_i' = D_i$, so $L_i^N \leq D_i'$.

HVH tasks: Virtual deadline $D'_i = D_i/(b-1) > D_i/b \geq L_i^N$ and $D_i - D'_i = (b-2)D_i/(b-1) > D_i/b \geq L_i^O$, since for any $b > 3$, we have $(b-2)/(b-1) > (b-2)/b \geq 1/b$.

HMH tasks: Virtual deadline is $D'_i = 2D_i/b > D_i/b \geq L_i^N$ and $D_i - D'_i = (b-2)D_i/b > D_i/b \geq L_i^O$ for $b > 3$. □

We now argue that a valid mapping S^T can be generated for typical-state for a capacity augmentation bound of $b = 2 + \sqrt{2}$.

Lemma 11 *If the Conditions of Definition 2 hold for any $b \geq 2 + \sqrt{2}$, then $m \geq N_{LH}^N + N_{HVH}^N + N_{HMH}^N$ and the typical state mapping S^T is valid.*

Proof **LHI tasks:** From Lemma 3, for any $b \geq 2 + \sqrt{2} > 3$ we have

$$N_{\mathrm{LH}}^N = \sum_{\tau_i \in \tau_{\mathrm{LH}}} n_i^N \leq \sum_{\tau_i \in \tau_{\mathrm{LH}}} (b-1)u_i^N = (b-1)U_{\mathrm{LH}}^N$$

HVH tasks: From Lemma 5, for any b we always have

$$N_{\mathrm{HVH}}^N = \sum_{\tau_i \in \tau_{\mathrm{HVH}}} n_i^N \leq \sum_{\tau_i \in \tau_{\mathrm{HVH}}} u_i^O = U_{\mathrm{HVH}}^O$$

HMH tasks: From Lemma 8, for $b \geq 2 + \sqrt{2} > 2$ we have

$$N_{\mathrm{HMH}}^N \leq \sum_{\tau_i \in \tau_{\mathrm{HMH}}} ((b-1)u_i^N + u_i^O) = (b-1)U_{\mathrm{HMH}}^N + U_{\mathrm{HMH}}^O$$

Since the Conditions of Definition 2 hold, we know $U^N = U_{\mathrm{LH}}^N + U_{\mathrm{HVH}}^N + U_{\mathrm{HMH}}^N \leq \frac{m}{b}$ and $U^O = U_{\mathrm{HVH}}^O + U_{\mathrm{HMH}}^O \leq \frac{m}{b}$. Therefore, we can derive

$$\begin{aligned}
&N_{\mathrm{LH}}^N + N_{\mathrm{HVH}}^N + N_{\mathrm{HMH}}^N \\
\leq &(b-1)U_{\mathrm{LH}}^N + U_{\mathrm{HVH}}^O + (b-1)U_{\mathrm{HMH}}^N + U_{\mathrm{HMH}}^O \\
\leq &(b-1)(U_{\mathrm{LH}}^N + U_{\mathrm{HVH}}^N + U_{\mathrm{HMH}}^N) + U_{\mathrm{HVH}}^O + U_{\mathrm{HMH}}^O \\
\leq &(b-1)m/b + m/b \\
= &m
\end{aligned}$$

Thus, typical-state mapping is always valid if $b = 2 + \sqrt{2}$. □

We now argue that the critical-state mapping S^C is valid for $b = 2 + \sqrt{2}$.

Lemma 12 *For a task set in critical-state under MCFS, if the Conditions of Definition 2 hold for $b = 2 + \sqrt{2} \approx 3.41$, then $m \geq N_{HVH}^O + N_{HMH}^O$ and mapping S^C is valid.*

Proof **HVH tasks:** From Lemma 6, for for $b = 2 + \sqrt{2} \approx 3.41 > \frac{7+\sqrt{33}}{4} \approx 3.19$

$$N_{\mathrm{HVH}}^O = \sum_{\tau_i \in \tau_{\mathrm{HVH}}} n_i^O \leq \sum_{\tau_i \in \tau_{\mathrm{HVH}}} bu_i^O = bU_{\mathrm{HVH}}^O$$

HMH tasks: From Lemma 9, for $b = 2 + \sqrt{2}$ we know

$$N^C_{\text{HMH}} = \sum_{\tau_i \in \tau_{\text{HMH}}} n^O_i \leq \sum_{\tau_i \in \tau_{\text{HMH}}} b u^O_i = b U^O_{\text{HMH}}$$

Since the Conditions of Definition 2 hold, $U^O = U^O_{\text{HVH}} + U^O_{\text{HMH}} \leq m/b$.

$$N^C_{\text{HVH}} + N^C_{\text{HMH}} \leq b U^O_{\text{HVH}} + b U^O_{\text{HMH}} \leq m$$

Thus, critical-state mapping is always valid if $b = 2 + \sqrt{2}$. □

Based on the properties of LH, HVH and HMH tasks, we observe that the bound of $2 + \sqrt{2}$ is only required for HMH tasks in the critical-state mapping.

5.6 Lower Bound on Capacity Augmentation for High-Utilization Tasks

Now we use an example task set to show the tightness of MCFS capacity augmentation bound.

Theorem 3 *The capacity augmentation bound for MCFS for high-utilization tasks on m cores is at least $\beta = 2 - \frac{3}{2m} + \sqrt{(2 - \frac{3}{2m})^2 - 2}$, when $m > 9$. When $m \to \infty$, $\beta \to 2 + \sqrt{2}$. When $m \to 9$, $\beta \to 3$.*

Proof On a system with m cores, consider a task set τ with a two tasks: τ_1 is a low criticality LH task with an utilization of $u_1 = 1 + \epsilon_1$, where ϵ_1 is an arbitrarily small positive number; τ_2 is a high criticality HMH task with an overload utilization of $u^O_2 = \frac{-3}{\beta^2 - 4\beta + 2}$ and a nominal utilization of $u^N_2 = (\frac{2}{b} - \frac{1}{\beta}) u^O_2$. In addition, the nominal and overload critical-path length of both tasks equals to $1/\beta$ of their deadlines.

Note that $\beta = 2 - \frac{3}{2m} + \sqrt{(2 - \frac{3}{2m})^2 - 2}$ is one of the roots of $\frac{-3}{\beta^2 - 4\beta + 2} = m/\beta$. Hence, the total overload utilization of the task set is $u^O_2 = m/\beta$, while the total nominal utilization of the task set is less than m/β. Therefore, the task set satisfies the conditions from Section 4.2 for a bound of β.

For HMH task τ_2, we know that by the MCFS mapping $b = 2 + \sqrt{2}$ and

$$\left\lceil \frac{C^N_2 - L^N_2}{2D_i/b - L^N_2} \right\rceil = \left\lceil \frac{\beta u^N_2 - 1}{2\beta/b - 1} \right\rceil = \left\lceil \frac{\beta(\frac{2}{b} - \frac{1}{\beta}) u^O_2 - 1}{2\beta/b - 1} \right\rceil < \lceil u^O_2 \rceil$$

Hence it is assigned with $n^N_2 = \lceil u^O_2 \rceil$ cores in the typical-state mapping.

Note that the larger the m, the larger the β. When $m \to \infty$, $\beta \to 2 + \sqrt{2}$. Hence, $\beta > b$. In the critical-state mapping, τ_2 is assigned with n^O_2 cores.

$$
\begin{aligned}
n_2^O &= \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil = \left\lceil \frac{\beta u_2^O - \frac{2\beta}{b}\left\lceil u_2^O \right\rceil - 1}{\beta - \frac{2\beta}{b} - 1} \right\rceil \\
&\geq \left\lceil \frac{\beta u_2^O - 2\left\lceil u_2^O \right\rceil - 1}{\beta - 2 - 1} \right\rceil > \left\lceil \frac{\beta u_2^O - 2(u_2^O + 1) - 1}{\beta - 2 - 1} \right\rceil \\
&= \left\lceil \frac{(\beta - 2)u_2^O - 3}{\beta - 3} \right\rceil \geq \frac{(\beta - 2)u_2^O - 3}{\beta - 3}
\end{aligned}
$$

Note that for $u_2^O = \frac{-3}{\beta^2 - 4\beta + 2}$, we have $\frac{(\beta-2)u_2^O - 3}{\beta - 3} = \beta u_2^O$. Hence, $n_2^O > \beta u_2^O = m$ and there is not enough cores to assign to task τ_2 in the critical-state mapping. Therefore, we reach the conclusion. □

6 MCFS for Multi-Criticality Systems

We now extend MCFS to systems with more than two criticality levels, still assuming that all tasks are high-utilization tasks. We describe the system model with 3 criticality levels, which can be generalized easily to more than three levels. Then, we will argue that MCFS provides a capacity augmentation bound of $(5 + \sqrt{5})/2$ for systems with 3 or more criticality levels.

6.1 Multi-Criticality System Model

The tuple $(Z_i, C_i^N, C_i^O, L_i^N, L_i^O, D_i)$ still represents a task; that is, in our model, each task still has two behaviours: nominal work C_i^N estimated by the system designer and overload work C_i^O estimated by the certification authorities (similarly for critical-path length).[1] In all the criticality levels that are same or below Z_i, task τ_i exhibits nominal behavior. If τ_i overruns its nominal parameters, then the system transitions to the criticality level Z_i. The only exception is for tasks with the lowest criticality level $Z_i = LO$: if they overrun their nominal parameters, they are allowed to miss their deadlines. An example for three criticality levels $Z_i \in \{\text{LO}, \text{ME}, \text{HI}\}$ for low, medium and high, is shown in Table 4.

A scheduler for a 3-Criticality System must satisfy the following conditions:

(1) If the system remains in the typical-state, then all tasks must meet their deadlines based on their nominal parameters (work and critical-path length);
(2) If any medium-criticality task exceeds its nominal parameters, then the system transitions into the intermediate-state — all medium- and high-criticality tasks must meet their deadlines based on their medium-criticality parameters (including work and critical-path length) shown in the second

[1] There is another multi-criticality model (Vestal, 2007) assuming that a task has more than two work estimates, one for each criticality level.

Table 4 Tasks' Per-Criticality Work, Critical-Path Length and Core Assignment of a 3-Criticality System

	Work, Critical-Path Length			Core Assignment under MCFS		
Task Criticality	LO-Work	ME-Work	HI-Work	Typical-State	Intermediate-State	Critical-State
LO	C_i^N, L_i^N	-	-	n_i^N	-	-
ME	C_i^N, L_i^N	C_i^O, L_i^O	-	n_i^N	n_i^O	-
HI	C_i^N, L_i^N	C_i^N, L_i^N	C_i^O, L_i^O	n_i^N	n_i^N	n_i^O

Table 5 High-Utilization Task Classification of a 3-Criticality System

Task Type	Criticality	Nominal Utilization	Overload Utilization
HMH	High	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$
HVH	High	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$
MMH	**Medium**	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$
MVH	**Medium**	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$
LH	Low	$1 < u_i^N$	NA

Table 6 High-Utilization Tasks' Assignments of a 3-Criticality System

Task Type	b	Virtual Deadline D_i'	Number of assigned cores n_i^N for nominal behavior	Number of assigned cores n_i^O for overload behavior
HMH	$2+\sqrt{2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D_i'-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil\right\}$
HVH	$2+\sqrt{2}$	$\frac{D_i}{b-1}$	$\lfloor u_i^O \rfloor$	$\left\lceil\frac{C_i^O-n_i^N D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil$
MMH	$\mathbf{(5+\sqrt{5})/2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D_i'-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil\right\}$
MVH	$\mathbf{(5+\sqrt{5})/2}$	$\frac{D_i}{b-1}$	$\lfloor u_i^O \rfloor$	$\left\lceil\frac{C_i^O-n_i^N D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil$
LH	2	D_i	$\left\lceil\frac{C_i^N-L_i^N}{D_i-L_i^N}\right\rceil$	NA

column in Table 4 (i.e. nominal parameters for high-criticality tasks and overload parameters for medium-criticality tasks). The scheduler is allowed to discard all low-criticality tasks;

(3) If any high-criticality task overruns its nominal parameters, then the system transition into the critical-state. High-criticality tasks still meet their deadlines based on their high-criticality parameters shown in the third column in Table 4 (i.e. overload parameters). The scheduler is allowed to discard all low and medium-criticality tasks.

6.2 Multi-Criticality MCFS Algorithm and Bound

We now generalize the MCFS algorithm to 3-criticality systems. The classification, virtual deadline and core assignments are shown in Table 5 and 6. Note that the classification is similar to the one shown in Section 4. Moreover, the assignments for medium-criticality tasks are almost identical to high-criticality tasks, except for a slightly larger b that is designed to provide the capacity augmentation bound of multi-criticality MCFS.

To calculate the mappings S^T, S^I, and S^C, we simply assign cores according to the task behavior shown in Table 4. For instance, in the intermediate-state mapping, a medium-criticality task gets n_i^O cores while a high-criticality task gets n_i^N cores. In the schedulability test, we add an additional condition saying that the total number of cores assigned in the intermediate-state is at most m. The other conditions remain the same. The following theorem gives the capacity augmentation bound.

Theorem 4 *Multi-criticality MCFS with only high-utilization tasks is correct and has a capacity bound of* $(5+\sqrt{5})/2$ *— if the conditions from Definition 2 hold for* $b = (5+\sqrt{5})/2$*, then the task set satisfies the following conditions of MCFS schedulability test:*

(1) virtual deadlines for high-utilization tasks are valid — any LH, HVH or HMH task has $L_i^N \le D_i'$ *and any HVH or HMH task has* $L_i^O \le D_i - D_i'$*;*
(2) typical-state mapping is valid — $n_{LH}^N + n_{MVH}^N + n_{MMH}^N + n_{HVH}^N + n_{HMH}^N \le m$*;*
(3) intermediate-state mapping is valid — $n_{MVH}^O + n_{MMH}^O + n_{HVH}^N + n_{HMH}^N \le m$*;*
(4) critical-state mapping is valid — $n_{HVH}^O + n_{HMH}^O \le m$*.*

Proof Recall that we noted in Section 5.4 that if the schedulability test passes for any $b > 2$, then the task set is schedulable. Since medium-criticality tasks are classified and scheduled just like high-criticality ones with the only modification that $b = (5+\sqrt{5})/2 \approx 3.62$, the correctness is obvious.

The proof of capacity augmentation is similar to Theorem 2: the critical-state mapping is identical to Lemma 12; and the proofs for the typical-state mapping and the virtual deadline are similar to Lemma 11 and Lemma 10, respectively, with the only difference being $b = (5+\sqrt{5})/2$.

Now we only prove the differences. We must bound the number of assigned cores in the intermediate-state mapping and prove that $N_{\mathrm{MVH}}^O + N_{\mathrm{MMH}}^O + N_{\mathrm{HVH}}^N + N_{\mathrm{HMH}}^N \le m$. Note that in the intermediate-state, medium-criticality tasks are assigned cores according to their overload parameters, while high-criticality tasks are assigned according to their nominal parameters.

For high-criticality tasks, from Lemmas 5 and 8 we have

$$N_{\mathrm{HVH}}^N + N_{\mathrm{HMH}}^N \le U_{\mathrm{HVH}}^O + (b-1)U_{\mathrm{HMH}}^N + U_{\mathrm{HMH}}^O$$

Medium-criticality tasks are more interesting. In order to use a lemma similar to Lemma 11, we must bound n_{MVH}^O and n_{MMH}^O. Unfortunately, it is not sufficient to bound them by bU_{MVH}^O and bU_{MMH}^O as in Lemmas 6 and 9.

Instead, we show a modified result similar to Lemma 6 and bound n^O_{MVH} by $(b-1)U^O_{\text{MVH}}$. From Lemma 6, we know that Inequality (3) is correct for any b. Since we want to bound n^O_{MVH} by $(b-1)U^O_{\text{MVH}}$, we need to solve the inequality $\frac{3b^2-7b+2}{2(b^2-3b+1)} \leq b-1$, which is equivalent to $2b^3-11b^2+15b-4 \geq 0$. We denote $f(b) = 2b^3-11b^2+15b-4$. Note that $f(b)$ is always increasing for $b \geq (5+\sqrt{5})/2 \approx 3.62 > (11+\sqrt{31})/6 \approx 2.76$ and $f((5+\sqrt{5})/2) = 1 > 0$. Therefore, for $b \geq (5+\sqrt{5})/2$, we have:

$$n^O_i \leq \frac{3b^2-7b+2}{2(b^2-3b+1)} u^O_i \leq (b-1)u^O_i$$

Similarly, we derive a modified result for Lemma 9. The Inequality (4) in Lemma 9 is correct for any $4 > b > 3$. Since we want to bound n^O_{MMH} by $(b-1)U^O_{\text{MMH}}$, this requires that $b-2 \leq (b-1)(b-3)$. Therefore, for $b \geq (5+\sqrt{5})/2 \approx 3.62$ we have

$$n^O_i \leq \frac{(b-2)u^O_i}{b-3} \leq (b-1)u^O_i$$

Thus, for $b \geq (5+\sqrt{5})/2$, we can show that $N^O_{\text{MVH}} \leq (b-1)U^O_{\text{MVH}}$ and $N^O_{\text{MMH}} \leq (b-1)U^O_{\text{MMH}}$. Therefore, the intermediate-state mapping has

$$\begin{aligned}
&n^O_{\text{MVH}} + n^O_{\text{MMH}} + n^N_{\text{HVH}} + n^N_{\text{HMH}} \\
\leq&(b-1)U^O_{\text{MVH}} + (b-1)U^O_{\text{MMH}} + U^O_{\text{HVH}} + (b-1)U^N_{\text{HMH}} + U^O_{\text{HMH}} \\
\leq&(b-1)(U^O_{\text{MVH}} + U^O_{\text{MMH}} + U^N_{\text{HVH}} + U^N_{\text{HMH}}) + U^O_{\text{HVH}} + U^O_{\text{HMH}} \\
\leq&(b-1)m/b + m/b \leq m
\end{aligned}$$

The typical-state mapping is similar to Lemma 11, while the critical-state mapping is the same as Lemma 12. By intersecting all the ranges of b, multi-criticality MCFS has a capacity bound of $(5+\sqrt{5})/2 \approx 3.62$. □

Remark: Why is the capacity augmentation bound $2+\sqrt{2}$ for dual-criticality systems and $(5+\sqrt{5})/2$ for systems with 3 or more criticality levels? In general, when a system is in the criticality level Z_i, the tasks at criticality level Z_i exhibit overload behavior, while all tasks with higher criticality levels exhibit nominal behavior. However, dual-criticality systems have a special property that lets us prove a better bound: when the system is in the typical-state, the low-criticality tasks exhibit nominal behavior (instead of overload behavior).

Generalization to more than 3 criticality levels: Consider the system with l criticality levels $\{\text{LO}, Z_2, ..., Z_i, ..., Z_{l-1}, \text{HI}\}$. The classification, virtual deadline and core assignments for a task with criticality Z_i where $1 < i < l$ are identical to medium-criticality tasks shown in Table 5 and 6, using $b = (5+\sqrt{5})/2$. In particular, tasks with criticality Z_i where $1 < i < l$ are classified into ZiMH and ZiVH tasks, based on their nominal utilization. Since the assignments for ZiMH tasks are the same as MMH tasks and the

assignments for ZiVH tasks are the same as MVH tasks, hence we can prove the correctness and capacity augmentation bound similarly.

Note that this bound is not related to the number of criticality levels. This is because in our model each task has two behaviours: the nominal behaviour estimated by the system designer and the overload behavior estimated by the certification authorities. For a task τ_i with criticality Z_i where $1 < i < l$, all tasks with lower criticality levels than τ_i are discarded; all tasks with higher but not highest criticality level Z_j where $i < j < l$ exhibit nominal behavior and have the same nominal core assignment using $b = (5+\sqrt{5})/2$, so we do not need to distinguish them. In addition, since Lemma 5 and 8 hold for $b = (5+\sqrt{5})/2$, for criticality level Z_j we have $n^N_{\text{ZjVH}} + n^N_{\text{ZjMH}} \leq U^O_{\text{ZjVH}} + (b-1)U^N_{\text{ZjMH}} + U^O_{\text{ZjMH}}$. Similar to (3) in Theorem 4, the mapping for criticality level Z_i is valid — $n^O_{\text{MVH}} + n^O_{\text{MMH}} + n^N_{\text{ZjVH}} + n^N_{\text{ZjMH}} + n^N_{\text{HVH}} + n^N_{\text{HMH}} \leq m$. Therefore, the capacity augmentation bound remains the same for more than 3 criticality levels.

7 Improved MCFS Algorithm for High-Utilization Tasks

In Sections 4 and 6, we presented the MCFS mapping algorithm for high-utilization tasks that are designed especially for task sets satisfying the conditions from Section 4.2 for a capacity augmentation bound of $2 + \sqrt{2}$ and $(5 + \sqrt{5})/2$ for dual- and multi-criticality task sets, respectively. Because of the design of this particular MCFS mapping algorithm, however, for task sets that violate these conditions it may not be able to find a valid mapping, even though there may exist a mapping that can schedule the task sets.

To improve the schedulability of MCFS algorithm, in this section we present the improved MCFS algorithm as ***MCFS-Improve*** based on heuristics for finding a valid mapping for task sets having even higher utilizations than is indicated by the capacity augmentation bound. For the rest of this section, we use the dual-criticality system as an example. The proposed MCFS-Improve algorithm can be easily extended from dual- to multi-criticality systems.

Why can't MCFS mapping algorithm generate a valid mapping for task sets that do not satisfy conditions of capacity augmentation bounds? This is mainly caused by the fixed parameter b in table 3 and 6. As discussed in Section 5.4, the correctness of MCFS does not rely on a particular b. The particular value of b is chosen to assign roughly b times the total utilization in nominal-state and also b times the total utilization in critical-state. When conditions of capacity augmentation bound of b are satisfied, the total utilizations in both states are less than m/b, so there are enough cores to assign to each task. However, when the conditions are not satisfied, e.g. the total utilization in critical-state is larger than m/b while the total utilization in typical-state is less than m/b, the available cores in the critical-state are not enough. Note that there are extra cores in the typical-state that are not assigned to any task. These cores could be assigned to some tasks in the typical-state, so that the tasks would need less cores in the critical-state.

1 // First, assign virtual deadlines and cores according to the basic MCFS mapping
2 $b = 2 + \sqrt{2}$
3 **for** each task τ_i in the task set
4 **if** τ_i is a LH task:
5 **if** $D_i - L_i^N \leq 0$: **return** unschedulable
6 $D'_i = D_i$; $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D_i - L_i^N} \right\rceil$; $n_i^O = 0$
7 **elseif** τ_i is a HVH task:
8 **if** $D_i - L_i^N - L_i^O \leq 0$: **return** unschedulable
9 **elseif** $\frac{b-2}{b-1} D_i > L_i^O$: $D'_i = \frac{D_i}{b-1}$; $n_i^N = \lfloor u_i^O \rfloor$; $n_i^O = \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil$
10 **else** : $D'_i = \frac{L_i^N D_i}{L_i^N + L_i^O}$; $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D'_i - L_i^N} \right\rceil$; $n_i^O = \max\left\{ n_i^N, \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil \right\}$
11 **elseif** τ_i is a HMH task:
12 **if** $D_i - L_i^N - L_i^O \leq 0$: **return** unschedulable
13 **elseif** $\frac{b-2}{b} D_i > L_i^O$: $D'_i = \frac{2}{b} D_i$; $n_i^N = \max\left\{ \left\lceil \frac{C_i^N - L_i^N}{D'_i - L_i^N} \right\rceil, \lceil u_i^O \rceil \right\}$;
14 **else** : $D'_i = \frac{L_i^N D_i}{L_i^N + L_i^O}$; $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D'_i - L_i^N} \right\rceil$;
15 $n_i^O = \max\left\{ n_i^N, \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil \right\}$

Fig. 1 MCFS-Improve mapping algorithm and schedulability test (initialization step)

The MCFS-Improve algorithm is designed based on this observation. As shown in Figure 1, it starts with the original MCFS mapping in table 3, with the exception for tasks with critical-path lengths longer than D_i/b. For these tasks, the originally assigned virtual deadline may not be enough to complete the critical-path length in one of the states. Unlike the basic MCFS that deems such tasks unschedulable, MCFS-Improve still tries to calculate a valid virtual deadline in the initialization step. If the initial core assignments in the typical- and critical-state are already valid, then MCFS-Improve admits this task set. Note that for task sets satisfying the conditions of capacity augmentation bound, the virtual deadline and core assignments are the same, so MCFS-Improve also has a capacity augmentation bound of $2 + \sqrt{2}$.

Moreover, for task sets without a valid mapping in the initialization step, MCFS-Improve adjusts the virtual deadline and core assignment according to some heuristics. At a high level, according to whether the number of cores is insufficient in the typical- or critical-state, MCFS-Improve selectively chooses some tasks, and decreases or increases the number of cores assigned in the typical-state. The tasks are chosen greedily, in order to have the maximum impact on balancing the core assignments in both states. For example, when there are not enough cores in the critical-state, MCFS-Improve chooses a task τ_i to increase the number of cores in the typical-state. With extra cores, τ_i is able to complete its nominal work faster, allowing a shorter virtual deadline and hence more time to complete its overload work. Thus, the number of cores required by τ_i in the critical-state could decrease. Task τ_i is selected, such that it has the maximum decrease of the number of cores in the critical-state. By

$n^N = \sum_i n_i^N$, $n^O = \sum_i n_i^O$
// Both critical- and typical-state mappings are valid
if $n^N \leq m$ and $n^O \leq m$: **return** schedulable
// Both critical- and typical-state mappings are not valid
elseif $n^N > m$ and $n^O > m$: **return** unschedulable
// There are not enough cores for critical-state mapping
elseif $n^N \leq m$ and $n^O > m$:
    while $n^N = \sum_i n_i^N \leq m$:
        $a = \min\{m - n^N, 1\}$
        for each task τ_i in the task set, where τ_i is not a LH task:
            $\bar{n}_i^N = n_i^N + a$, $\bar{D}'_i = \frac{C_i^N - L_i^N}{\bar{n}_i^N} + L_i^N$;
            if $\bar{n}_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O}$: $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N \bar{D}'_i - L_i^O}{D_i - \bar{D}'_i - L_i^O} \right\rceil$
            else : $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N (\bar{D}'_i - L_i^N) - L_i^O}{D_i - \bar{D}'_i - (L_i^O - L_i^N)} \right\rceil$
            $x_i = n_i^O - \bar{n}_i^O$
        τ_i is the task with the maximum decrease x_i
        for τ_i, update $D'_i = \bar{D}'_i$, $n_i^N = \bar{n}_i^N$, $n_i^O = \bar{n}_i^O$
        if $a == 0$: for each τ_i, update $D'_i = \bar{D}'_i$, $n_i^N = \bar{n}_i^N$; $n_i^O = \bar{n}_i^O$; **break**
    if $n^O = \sum_i n_i^O \leq m$: **return** schedulable
    else : **return** unschedulable
// There are not enough cores for typical-state mapping
elseif $n^N > m$ and $n^O \leq m$:
    while $n^O = \sum_i n_i^O \leq m$:
        $b = \min\{m - n^O, 1\}$
        for each task τ_i in the task set, where τ_i is not a LH task:
            $\bar{n}_i^N = n_i^N - 1$, $\bar{D}'_i = \frac{C_i^N - L_i^N}{\bar{n}_i^N} + L_i^N$;
            if $D_i - \bar{D}'_i - L_i^O \leq 0$: $y_i =$MAX, **continue**
            if $\bar{n}_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O}$: $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N \bar{D}'_i - L_i^O}{D_i - \bar{D}'_i - L_i^O} \right\rceil$
            else : $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N (\bar{D}'_i - L_i^N) - L_i^O}{D_i - \bar{D}'_i - (L_i^O - L_i^N)} \right\rceil$
            $y_i = \bar{n}_i^O - n_i^O$
        τ_i is the task with the minimum increase y_i
        if $y_i \leq b$: for τ_i, update $D'_i = \bar{D}'_i$, $n_i^N = \bar{n}_i^N$, $n_i^O = \bar{n}_i^O$
        else : **break**
    if $n^N = \sum_i n_i^N \leq m$: **return** schedulable
    else : **return** unschedulable

Fig. 2 MCFS-Improve mapping algorithm and schedulability test (adjustment step)

increasing the total cores assigned in typical-state, the total cores assigned in critical-state could eventually be less than m. The adjustment strategy is similar for the case where there are not enough cores in the typical-state.

The correctness of MCFS-Improve algorithm in Figure 2 can be derived from the following lemma, which tells us how to calculate the minimum number of cores required in the critical-state, given the core assignment in the typical-state. Note that for LH tasks, the core assignment is already tight for the typical-state and the LH tasks are not assigned any core in the critical-state, so MCFS-Improve does not change the mapping for LH tasks.

Lemma 13 *If a high-criticality high-utilization task τ_i is assigned with n_i^N cores in the typical-state, then it can complete its nominal work by a virtual deadline of D'_i, where $D'_i = \frac{C_i^N - L_i^N}{n_i^N} + L_i^N$. Given n_i^N and D'_i, it only needs n_i^O cores to complete the overload work by deadline D_i, where*

$$n_i^O = \begin{cases} \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil & n_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O} \\ \left\lceil \frac{C_i^O - n_i^N (D'_i - L_i^N) - L_i^O}{(D_i - D'_i) - (L_i^O - L_i^N)} \right\rceil & n_i^N \geq \frac{C_i^O - L_i^O}{D_i - L_i^O}. \end{cases}$$

Proof We can easily prove that the virtual deadline is sufficient by applying Lemma 1. For the number of cores assigned in the critical-state, the proof is similar to that of Lemma 7. Let $t^* \leq D'_i$ be the number of complete steps where all cores are busy working and $D'_i - t^* \leq L_i^N$ be the number of incomplete steps where the critical-path length decreases before the transition. Then, at the transition, the job has $C^O - n_i^N t^* - D'_i + t^*$ remaining work and $L_i^O - D'_i + t^*$ remaining critical-path length that must be completed in $D_i - D'_i$ time steps. Therefore, τ_i is guaranteed to complete by the deadline, if τ_i is allocated at least n dedicated cores, where

$$n = \left\lceil \frac{(C^O - n_i^N t^* - D'_i + t^*) - (L_i^O - D'_i + t^*)}{(D_i - D'_i) - (L_i^O - D'_i + t^*)} \right\rceil = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil$$

Now consider two cases:
Case 1: If $n_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O}$, by applying Ineq (1) and $t^* \leq D'_i$, we get

$$n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil \leq \left\lceil \frac{C_i^O - L_i^O - n_i^N D'_i}{D_i - L_i^O - D'_i} \right\rceil = n_i^O$$

Case 2: If $n_i^N \geq \frac{C_i^O - L_i^O}{D_i - L_i^O}$, by applying Ineq (2) and $t^* \geq D'_i - L_i^N$, we get

$$n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil \leq \left\lceil \frac{C_i^O - L_i^O - n_i^N (D'_i - L_i^N)}{D_i - L_i^O - (D'_i - L_i^N)} \right\rceil = n_i^O$$

Combining the two cases gives us the proof. □

Remark: MCFS-Improve algorithm is a greedy algorithm that runs in polynomial time. The numerical experiments conducted in Section 10 are consistent with the analysis that MCFS-Improve can admit more task sets than the original MCFS algorithm and it can admit task sets with utilizations much higher than that indicated by the capacity augmentation bound. Note that due to the integer requirement of core assignment as well as the complexity of the calculation, the derived mapping may not be the global optimal mapping. However, when $C_i^O - L_i^O > C_i^N - L_i^N$ and $(C_i^O - L_i^O)(D_i - L_i^O - L_i^N) > (C_i^N - L_i^N)L_i^N$ (both are reasonable constraints on a parallel task) the calculation is approximately convex, so the algorithm may find the close to optimal mapping.

8 General Case for Dual-Criticality MCFS

We can generalize MCFS to dual-criticality task systems with both high- and low-utilization tasks (both the nominal utilization and overload utilization are at most 1). The high-utilization tasks are still scheduled in a similar manner as in Section 4, while we treat low-utilization tasks as sequential tasks and schedule them using a mixed-criticality multiprocessor scheduler for sequential tasks. In particular, in addition to the three categories from Section 4, we have two additional categories specific to low-utilization tasks. **LO-Low (LL)** tasks are low-criticality tasks with low-utilization in nominal behavior, i.e. $u_i^N \leq 1$. **HI-Low-Low (HLL)** tasks are high-criticality tasks with low-utilization in both behaviors, i.e. $u_i^N \leq u_i^O \leq 1$. We denote the set of low-utilization tasks as $\tau_{\text{Seq}} = \{\tau_i \in \tau_{\text{LL}} \cup \tau_{\text{HLL}}\}$ and their total utilizations in nominal and overload behavior as U_{Seq}^N and U_{Seq}^O, respectively.

Note that these tasks are essentially sequential tasks, since they do not require parallel execution to meet their deadlines in either states. Therefore, MCFS can use any existing mixed-criticality multiprocessor scheduler $\mathcal{S}$ for sequential tasks to schedule these tasks. Here, as an example, we assume that we will use MC-Partition (Baruah et al, 2014) to assign these tasks to cores. Say that the total numbers of assigned cores in typical- and critical-state to these tasks is N_{Seq}^N and N_{Seq}^O; unlike MC-Partition, these may be unequal.

At runtime, in the typical-state, high-utilization tasks still execute on their dedicated cores in parallel, while all tasks in τ_{Seq} execute on shared N_{Seq}^N cores. If any HLL task overruns its nominal work, the system transitions to critical-state and *all* LL tasks are immediately discarded. In addition, if $N_{\text{Seq}}^O > N_{\text{Seq}}^N$, some HLL tasks may need to migrate to cores assigned to some additional low-criticality tasks (from set LH). LH tasks may also be discarded in order to acquire N_{Seq}^O total cores for HLL tasks in overload behavior.

Therefore, even though high-utilization tasks never migrate even when the system transitions to critical-state, migration may be required for low-utilization tasks. This is because two or more HLL tasks may share a core in typical state without exceeding the nominal utilization bound of a single core, but their total overload utilization may increase to more than the capacity of a single core. For example, if we choose to use the MC-Partition-UT-0.75 algorithm presented in (Baruah et al, 2014) to schedule low-utilization tasks, the total numbers of assigned cores N_{Seq}^N and N_{Seq}^O in typical- and critical-state could be different. In such cases, some high-criticality low-utilization tasks may migrate when the system transitions from typical- to critical-state.

Correctness and Capacity Augmentation Bound: Recall that tasks in the LL and HLL categories are scheduled using the chosen scheduler $\mathcal{S}$ and are executed sequentially since they have utilization no more than 1 in both states. Therefore, the correctness of this algorithm follows from Section 5.4 and from the correctness of the chosen scheduler $\mathcal{S}$ for low-criticality tasks.

The following theorem proves the capacity augmentation bound.

Theorem 5 *For dual-criticality systems with both high- and low-utilization tasks, MCFS has a capacity augmentation bound of* $(s+1)m/(m-1)$ *where* $\frac{1}{s} \geq \frac{1}{1+\sqrt{2}}$ *is the utilization bound of the mixed-criticality multiprocessor scheduler* $\mathcal{S}$ *used by MCFS to schedule low-utilization tasks. For instance using a modified version of MC-partition (Baruah et al, 2014), we get a bound of* $11/3 \times m/(m-1) \approx 3.67$ *for large* m.

Proof The proof for capacity augmentation is obtained by simply noticing that all the relevant lemmas for high-utilization tasks, namely Lemmas 3, 5, 8, 6, and 9, work for $s + 1 \geq 2 + \sqrt{2}$. In addition, Section 4 shows that virtual deadlines are valid for any $b > 3$. Here we denote $U^N = U^N_{\text{Seq}} + U^N_{\text{LHi}} + U^N_{\text{HVH}} + U^N_{\text{HMH}} \leq m/b$ and $U^O = U^O_{\text{Seq}} + U^O_{\text{HVH}} + U^O_{\text{HMH}} \leq m/b$.

As $\frac{1}{s}$ is the utilization bound of the chosen scheduler $\mathcal{S}$ for low-utilization tasks, the number of cores assigned to them in nominal and critical-states are

$$N^N_{\text{Seq}} = \lceil sU^N_{\text{Seq}} \rceil < sU^N_{\text{Seq}} + 1 \text{ and } N^O_{\text{Seq}} = \lceil sU^O_{\text{Seq}} \rceil < sU^N_{\text{Seq}} + 1$$

As $b = (s+1)/(1-1/m)$, for typical-state mapping we can derive

$$\begin{aligned} & N^N_{\text{Seq}} + N^N_{\text{LHi}} + N^N_{\text{HVH}} + N^N_{\text{HMH}} \\ \leq & sU^N_{\text{Seq}} + 1 + sU^N_{\text{LHi}} + U^O_{\text{HVH}} + sU^N_{\text{HMH}} + U^O_{\text{HMH}} \\ \leq & sU^N + 1 + U^O \leq sm/b + 1 + U^O \\ \leq & \left((1-1/m)b - 1\right) m/b + 1 + m/b = m \end{aligned}$$

For critical-state mapping, we can also derive that

$$N^O_{\text{Seq}} + N^O_{\text{HVH}} + N^O_{\text{HMH}} \leq sU^O + 1 \leq sm/b + 1 \leq m$$

For instance, since MC-Partition-UT-0.75 has a utilization bound[2] of $\frac{3}{8}$, MCFS using MC-Partition has a capacity augmentation bound approaching $\frac{11}{3}$, when m is large and $m/(m-1) \approx 1$. □

In principle, the *MCFS scheduler*, can handle low-utilization tasks by utilizing any mixed-criticality multiprocessor scheduling strategy for sequential tasks. However, as far as we know, there is no prior work that shows a utilization bound for systems with more than 2 criticality levels; therefore, for such task sets we cannot prove a capacity-augmentation bound and restrict ourselves to tasks which exhibit high-utilization at least in their overload state.

9 Implementation of a MCFS Runtime System

We demonstrate the applicability of MCFS, as described in Section 4, by implementing a real-time MCFS runtime system for a dual-criticality system with high-utilization tasks. This reference implementation supports parallel programs written in OpenMP (OpenMP, 2013). It uses Linux with the

[2] As proved in (Baruah et al, 2014), it has a speedup of $\frac{8}{3}$ compared to 100% utilization.

RT_PREEMPT patch as the underlying RTOS and the OpenMP parallel concurrency platform to manage threads and assign work at runtime.

Three key requirements are derived for the MCFS runtime: (1) the system must detect when any high-criticality task has overrun its virtual deadline; (2) it must modify the core allocation to give more cores to high-criticality tasks in the event of a virtual deadline miss; and (3) since the number of active threads in the system fluctuates with its criticality state, it must provide a state-aware concurrency mechanism to facilitate parallel programming — i.e., a state-aware barrier.

Overrun Detection: The MCFS runtime system detects that a high-criticality task overruns its virtual deadline via Linux's `timer_create` and `timer_settime` API. These timers are set and disarmed at the start and end of each period by each high-criticality task while in the typical-state, so expiration only occurs in the event of an overrun. Timer expirations are delivered via signals and signal handlers. To make sure that the timer expiration is noticed promptly, kernel `ksoftirq` threads are given higher real-time priority than all other threads.[3]

Core Reallocation: A key requirement of MCFS is to increase the allocation of cores to a high-criticality task when it exceeds its virtual deadline, by taking cores away from low-criticality tasks. This is accomplished in four parts. (1) At the start of execution, each high-criticality task τ_i creates the maximum number of threads it would need in the critical-state (n_i^O). Each low criticality task creates n_i^N threads. (2) When the runtime system initializes (in typical-state), only n_i^N threads are awake for each task and they are pinned to distinct cores[4]. (3) The remaining $n_i^O - n_i^N$ threads of high-criticality tasks are put to sleep with the `FUTEX_WAIT` system call, while also pinned to their cores (which may be shared with a low-criticality task). These threads sleep at a priority higher than any low-criticality thread on the same core. (4) When a job of high-criticality task τ_i overruns its virtual deadline, its sleeping threads are awoken with `FUTEX_WAKE` and they preempt the low-criticality thread on the same core and begin executing.

Note that the set of cores assigned by the typical-state mapping to τ_i is a subset of the cores assigned by the critical-state mapping; therefore, the system needs no migration for the high-utilization tasks.

In this design, the threads of each task must be activated and deactivated each period via the OpenMP directive `#pragma omp parallel`. Thus, this approach of maintaining a pool of unused, high-criticality threads does impose an additional overhead on the system, even if it never transitions into critical-state, due to these activations and deactivations. However, these overheads are

[3] This could be a potential source of criticality inversion; however, in our system, this is not a major source of overhead. The alternative, thread-context notification, can be subject to unsuitably long delays.

[4] In order to pin threads to cores, before the task execution, we use an initial `#pragma omp parallel` directive where individual threads make a call to Linux's `sched_setaffinity` and pin themselves to the assigned cores.

only imposed on low-criticality tasks by high-criticality tasks, so there is no criticality inversion.

When a job of high-criticality task τ_i overruns its virtual deadline and preempts the low-criticality tasks on the shared cores, the current jobs of these low-criticality tasks may continue to execute when the higher-priority threads from high-criticality tasks are idling. If, however, the start times of the these low-criticality jobs are already later than their absolute deadlines, such jobs are dropped voluntarily by low-criticality tasks. Therefore, when the system is able to recover from critical-state to typical-state, there is little backlog of low-criticality jobs and the future arriving jobs of the same task are able to resume normal execution. Note that for systems that can tolerate tardiness for low-criticality jobs, an alternative implementation would not drop these backlogged jobs.

The primary reason for allowing current low-criticality jobs to run at a lower priority instead of directly killing the threads of these job is to avoid the cost of creating new threads during system operation, but it also allows the low-criticality threads to make progress on a best-effort basis. Note that since we allow low-criticality threads to continue executing after a mode transition has occurred, they will continue to interfere with high-criticality threads through cache pollution, resource contention, and other effects. Even so, allowing low-criticality threads to continue progressing seems appropriate for a soft real-time system. The other options are to kill these processes or to suspend them, but we do not investigate these options here.

Since high-criticality tasks do not share cores in MCFS, if a high-criticality task receives a timer signal, indicating that it has overrun its virtual deadline, it does not initiate a system-wide mode switch. Instead, it simply wakes up its sleeping $n_i^O - n_i^N$ threads and acquires the necessary additional cores from a subset of low-criticality tasks. If a low-criticality task overruns its deadline, it need not do anything. This natural implementation leads to graceful degradation since not all low-criticality tasks are discarded on entering critical-state.

Latency due to mode transition: The most important factor to optimize for ensuring the safe operation of high-criticality tasks is the *high-criticality activation latency*— the delay between when a mode transition is detected and when the additional $n_i^O - n_i^N$ high-criticality threads that were sleeping in the typical mode wake up and are ready to perform work. We measure this by inducing a mode transition at a fixed time, and the extra threads perform a time-stamp as soon as they wake up. The difference between the mode switch time and the latest time-stamp gives the latency. This latency was very low in general and increases with the increasing number of threads. We varied the number of awoken threads from one to fourteen, measuring the latency 400 times for each setting, and the maximum observed latency was 84 microseconds.

Note that this mode transition latency may occur only once for each high-criticality job in the critical-state. To incorporate it into schedulability analysis, we subtract it from the deadline of each high-criticality task.

Impact of high-criticality tasks on low-criticality tasks: As discussed in Section 9, low-criticality tasks incur overhead when they share a core with a high-criticality task. The low-criticality task is subject to interruption by high-criticality threads that must sleep and awake at the start and end of every period, which involves two context switches, the start and end of a `#pragma omp parallel` directive, and interactions with a Linux futex. We compare the wall-clock execution time of the low-criticality task with the Linux clock source `CLOCK_THREAD_CPUTIME_ID` to infer the total amount of time the low-criticality task was preempted. The maximum observed overhead was relatively high at 1555 microseconds per preemption. In our system, it was important to incorporate this overhead into the schedulability test to ensure that low-criticality tasks meet their deadlines. This overhead is only incurred when a high-criticality task's sleeping thread is sharing a core with a low-criticality task in the typical-state. In addition, the preemption only occurs once per period of the high-criticality task. Therefore, we can calculate the maximum number of preemptions and subtract the appropriate time from the low-criticality task's deadline. This allows the scheduler to assign the correct number of cores to low-criticality tasks.

Discussion: For tasks in our experiments on the simple prototype platform, we were able to mitigate the effect of this overhead by incorporating it into the schedulability test. For tasks at smaller time scales, this overhead may be unacceptably high. It is mostly attributed to the cost of entering and exiting the `#pragma omp parallel` each period as shown in Figure 3. For a reference system like we have described here, the choice of including the `parallel` directive within the periodic invocation greatly simplifies programming and reasoning about the system, as well as allows the user to use existing parallel programs with little modification, but the overhead may be unsuitably high for practical systems. In a traditional OpenMP program, the `parallel` directive would be used once or just a few times— calling it once every period exposes an important limitation of this standard parallel concurrency platform when used in real-time systems.

State-Aware Barrier Implementation: One side-effect of mixed-criticality model for parallel tasks is that counting-based thread synchronization methods such as barriers will not work properly as the number of active threads fluctuates. For OpenMP in particular, if some threads in an OpenMP team are sleeping (as in our implementation), the implicit barrier at the end of each `#pragma omp for` loop may deadlock, if the sleeping threads never arrive.

We address this by removing the implicit barrier with the OpenMP clause `nowait`, as shown in Figure 3 and implementing a state-aware barrier shown in Figure 4, which operates as follows. When a task begins a transition, its signal handler sets a variable indicating that the barrier needs updating before waking the extra high-criticality threads. The next thread to encounter the barrier checks this variable and claims responsibility for updating with an atomic compare-and-swap on a boolean flag. Other threads arriving after that will spin-wait. The update thread will then verify that the barrier is

```
1  perioidic_iteration(){
2        #pragma omp parallel
3        {
4        if(typical_state && high_crit_task)
5              sleep_extra_threads()
6
7        // Do parallel program
8        #pragma omp for schedule(dynamic) nowait
9        for (j = 0; j < num_strands; ++j)
10       {
11             // Perform work
12             busy_work()
13       }
14
15       mc_barrier_wait()
16       wake_extra_threads()
17       }
18 }
```

Fig. 3 Periodic Task Invocation Psuedocode

```
1  // Called asynchronously by signal handler
2  barrier_state_switch()
3        needs_switch = true
4
5  check_needs_updating()
6        if ( needs_switch )
7              atomically_claim_switcher()
8              if ( switcher )
9                    verify_barrier_inactive()
10                   update_barrier_count()
11                   needs_switch = false
12                   release_spinwaiters()
13             else spinwait()
14
15 mc_barrier_wait()
16       check_needs_updating()
17       do_barrier_wait()
```

Fig. 4 Mode Aware Barrier Psuedocode

not currently being modified by any thread that arrived before the transition, spin-waiting otherwise, and finally will increment the barrier count when it is safe to do so. It then releases any threads that are spin-waiting so that they may proceed through the barrier.

This imposes a small, constant overhead every time a thread accesses the barrier, since threads must check to see if the barrier needs updating. However, it allows us to use the same barrier in both states, and the barrier can be updated even if some threads are currently waiting on the barrier. Without such an arrangement, the transition overhead could be unbounded, since the

additional $n_i^O - n_i^N$ high-criticality threads could not be released while any barrier was in an indeterminate state.

Recover from critical-state to typical-state: As observed at the end of Section 4, the MCFS scheduling theory naturally supports tasks that may transition between the typical-state and critical-state many times over the life of the system. This is desirable as it allows low-criticality tasks to continue executing on a best-effort basis. Otherwise, a high-criticality task transitioning into critical-state would permanently impair any low-criticality task it happened to share a processor core with, even if the conditions that lead to the state transition were transient.

Reverting to typical-state is straightforward compared with transitioning into the critical-state, because the MCFS theory allows this to happen at a time of our choosing and not in response to any external event. Thus, a particularly convenient time for this to occur is outside the execution of any job of the task, because the task's team of parallel threads is not active during those times. Modifying the system while a parallel computation is underway is the major source of complexity for the critical-state transition and is what requires the complex core reallocation and state-aware barrier mechanisms that are discussed above.

Affecting the transition to typical-state requires resetting the state-aware barrier and reducing the number of threads that will participate in future job invocations. Since this process occurs outside the execution of any job, it is guaranteed that the barrier is not in use and that no parallel threads are active. Thus there are no concurrency issues to resolve, and reversion is accomplished without synchronization. In particular, the state-aware barrier is reconfigured to expect the number of threads that should be active in the typical-state (i.e. a modified version of `update_barrier_count()` from Figure 4 may be called without protection). Second, a global flag is set which indicates to the critical-state threads that they should sleep with `FUTEX_WAIT` upon activation rather than immediately participating.

Under the MCFS theory this reversion may be performed as often as the finish of each individual job that has entered the critical-state. In effect, the critical-state transition occurs on a per-job basis rather than a per-task basis, and all new jobs start in the typical-state but may transition to the critical-state as needed, allowing for very fine grained control over the system criticality and providing the minimum interruption to low-criticality tasks. Such low-criticality tasks operate on a best-effort basis but are not guaranteed in the face of interference from a task in the critical-state. In Section 10, we construct benchmark task set to test and evaluate the recovery to typical-state feature of MCFS runtime system.

10 Numerical Evaluation

We first conduct an extensive study comparing the schedulability tests of basic MCFS, MCFS-Improve and capacity augmentation bound of $2+\sqrt{2}$ on a wide

range of parameters. In particular, we investigate the impact of the following parameters on the schedulabilities of MCFS and MCFS-Improve: the number of cores m, the total nominal utilization U^N, the total overload utilization U^O, and the maximum ratio p_{max} of the overload critical-path length over period. For each of these settings, we ran the MCFS schedulability test on 1000 task sets and show the fraction of schedulable task sets admitted by MCFS and MCFS-Improve.

10.1 Task Set Generation

First, we explain how we generate task sets for the evaluation. We vary the number of cores m from 16, 32, 64 and 128. Given m cores, we vary both the total nominal utilization U^N and overload utilization U^O of task sets from 1 to m.

With the desired total nominal and overload utilization U^N and U^O of each setting, we first add high-criticality tasks until U^O is reached and then add low-criticality tasks until U^N is reached. To limit high-criticality tasks' total nominal utilization to U^N, for each setting we calculated a maximum "nominal over overload utilization ratio" $r_{max} = U^N/U^O$. High-criticality tasks' nominal over overload utilizations will not exceed the maximum ratio.

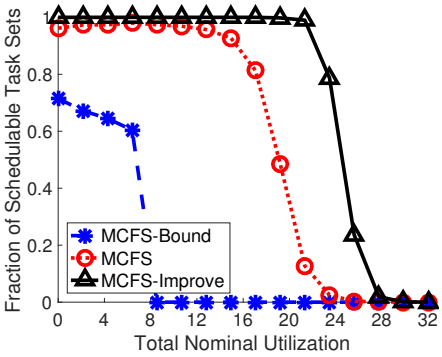
To evaluate the impact of critical-path length, we also vary the maximum ratio p_{max} of the overload critical-path length over period as $p_{max} = [\frac{0.25}{b}, \frac{0.5}{b}, \frac{0.75}{b}, \frac{1}{b}, \frac{1.25}{b}, \frac{1.5}{b}, \frac{1.75}{b}, \frac{2}{b}, \frac{2.25}{b}, \frac{2.5}{b}, \frac{2.75}{b}, \frac{3}{b}]$, where $b = 2+\sqrt{2}$. Note that the capacity augmentation bound of b requires that $p_{max} \leq \frac{1}{b}$. The larger the p_{max}, the harder to schedule the task set under the family of MCFS algorithms.

Note that for the two MCFS algorithms, the schedulability only depends on the work and critical-path length of tasks rather than their parallel structures (whether they are synchronous or DAG tasks). Therefore, for the numerical experiments we directly generate these parameters for each task.

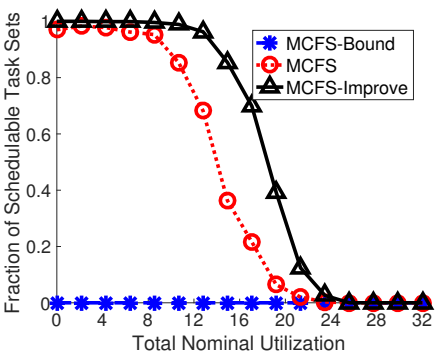
Our task set generation proceeds as follows.

(1) Criticality z_i: 50% high-criticality and 50% low-criticality.
(2) Nominal and overload utilization ratio r_i for high-criticality task: uniformly from 0.01 and a calculate r_{max}; This ratio r_i for low-criticality task is always 1.
(3) Overload utilization u_i^O: randomly chosen from a log normal distribution with mean of $1 + \sqrt{m}/3$.
(4) Nominal utilization $u_i^N = r_i u_i^O$.
(5) Implicit deadline D_i: uniformly from 100ms to 1000ms.
(6) Max overload critical-path length L': 40%, 50%, 70% and 100% of $D_i p_{max}$, with probability of 0.4, 0.3, 0.2 and 0.1.
(7) Overload critical-path length L_i^O: uniformly chosen from $[0, L']$.
(8) Nominal critical-path length $L_i^N = r_i L_i^O$.

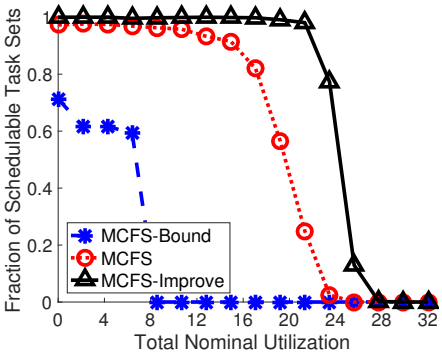
With the above parameters, we can calculate the nominal and overload work, which are used in the schedulability tests.



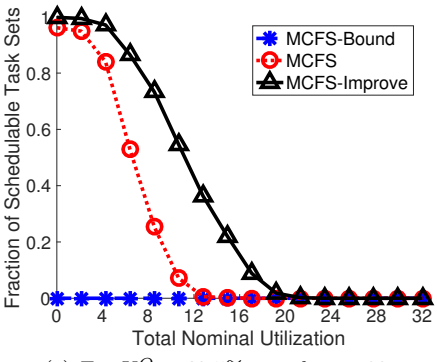
(a) For $U^O = 12.5\%m$ and $m = 32$.



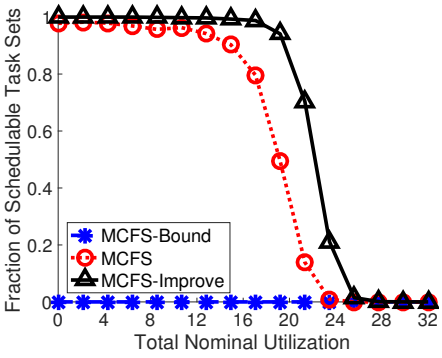
(d) For $U^O = 50\%m$ and $m = 32$.



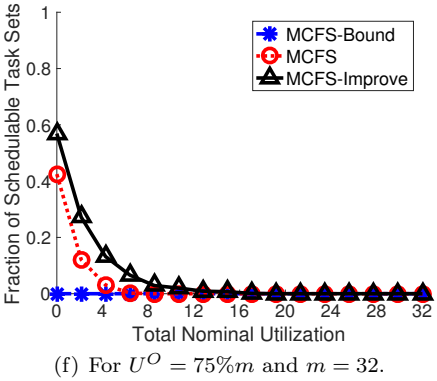
(b) For $U^O = 25\%m$ and $m = 32$.



(e) For $U^O = 62.5\%m$ and $m = 32$.

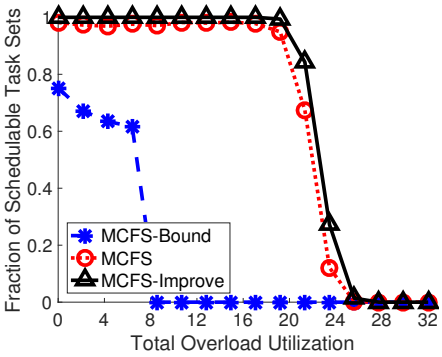


(c) For $U^O = 37.5\%m$ and $m = 32$.

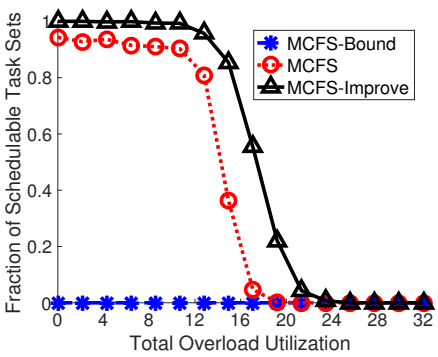


(f) For $U^O = 75\%m$ and $m = 32$.

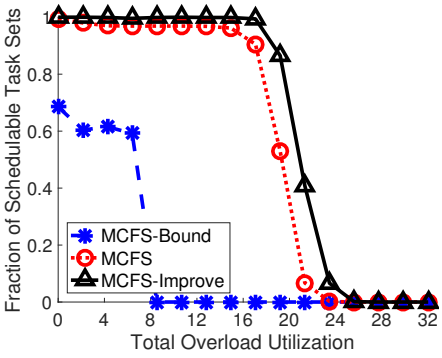
Fig. 5 Fraction of schedulable task sets of MCFS-Bound vs. MCFS vs. MCFS-Improve on ***32 cores*** setting with $\mathbf{p_{max}} = \frac{\mathbf{1.5}}{\mathbf{b}}$. From (a) to (f), figures show the results for total ***nominal utilizations*** ranging from $[12.5\%m, 25\%m, 37.5\%m, 50\%m, 62.5\%m, 75\%m]$, respectively. Each figure shows the results for increasing total ***overload utilizations***.



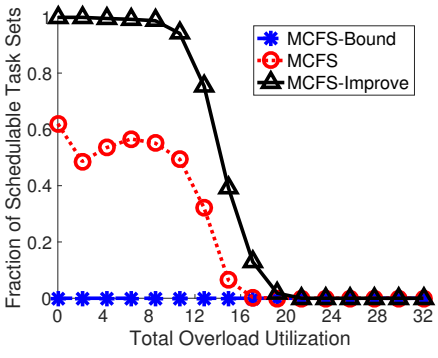
(a) For $U^N = 12.5\%m$ and $m = 32$.



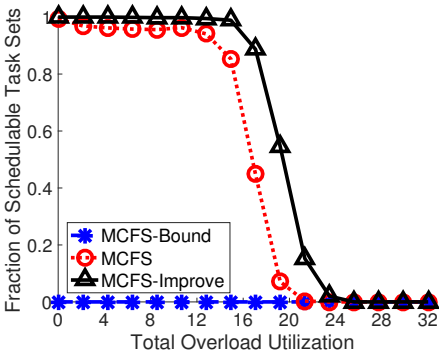
(d) For $U^N = 50\%m$ and $m = 32$.



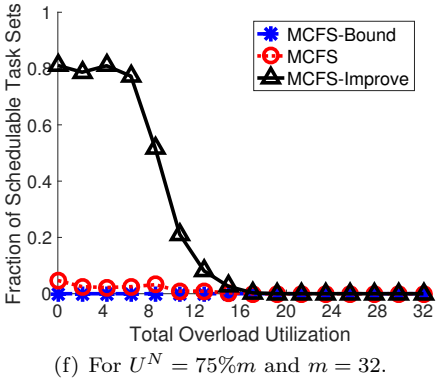
(b) For $U^N = 25\%m$ and $m = 32$.



(e) For $U^N = 62.5\%m$ and $m = 32$.

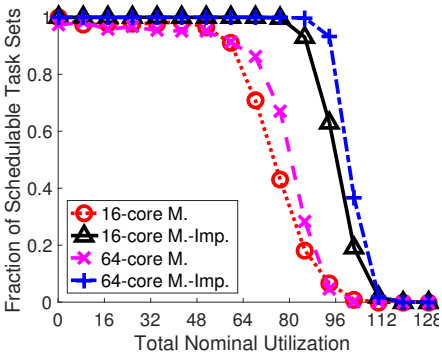


(c) For $U^N = 37.5\%m$ and $m = 32$.

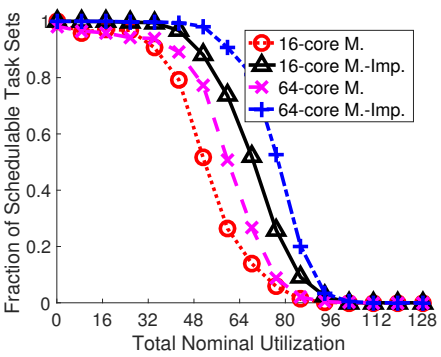


(f) For $U^N = 75\%m$ and $m = 32$.

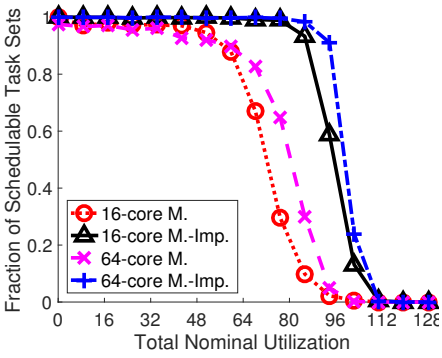
Fig. 6 Fraction of schedulable task sets of MCFS-Bound vs. MCFS vs. MCFS-Improve on ***32 cores*** setting with $\mathbf{p_{max}} = \frac{\mathbf{1.5}}{\mathbf{b}}$. From (a) to (f), figures show the results for total ***overload utilizations*** ranging from $[12.5\%m, 25\%m, 37.5\%m, 50\%m, 62.5\%m, 75\%m]$, respectively. Each figure shows the results for increasing total ***nominal utilizations***.



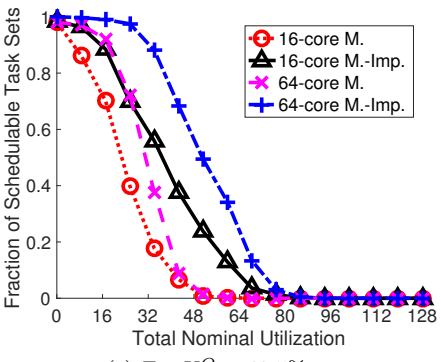
(a) For $U^O = 12.5\%m$.



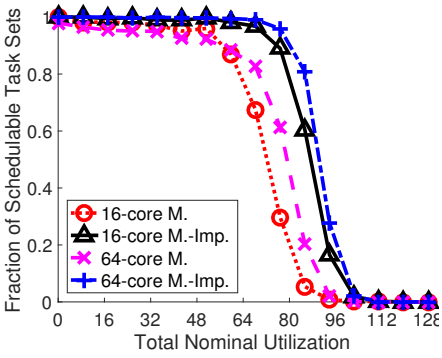
(d) For $U^O = 50\%m$.



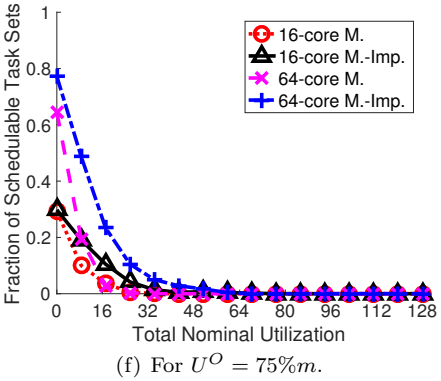
(b) For $U^O = 25\%m$.



(e) For $U^O = 62.5\%m$.

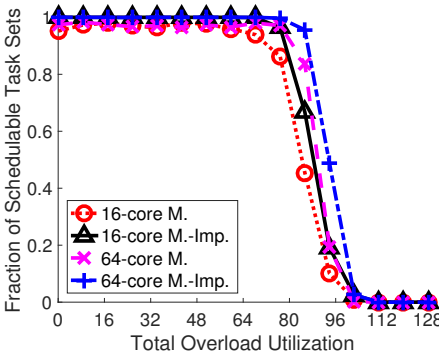


(c) For $U^O = 37.5\%m$.

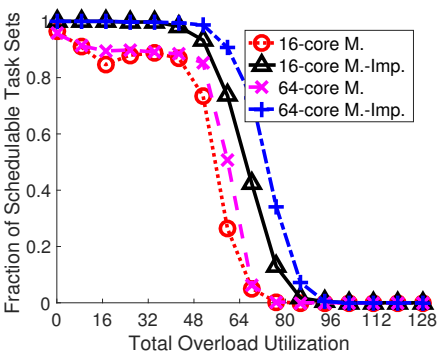


(f) For $U^O = 75\%m$.

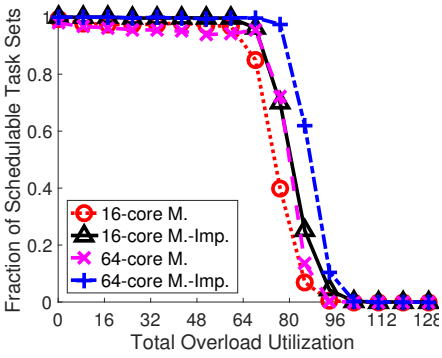
Fig. 7 Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on ***16 cores*** (labeled "16-core") vs. ***64 cores*** (labeled "64-core") with $\mathbf{p_{max}} = \frac{\mathbf{1.5}}{\mathbf{b}}$. From (a) to (f), figures show the results for varying total ***overload utilizations***. Each figure shows the results for increasing total ***nominal utilizations***.



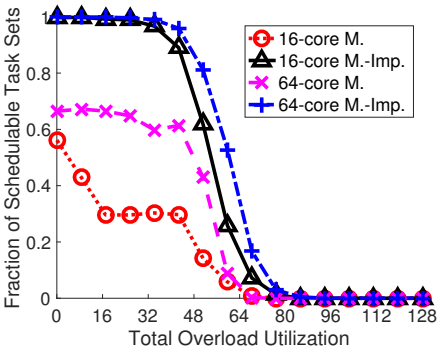
(a) For $U^N = 12.5\%m$.



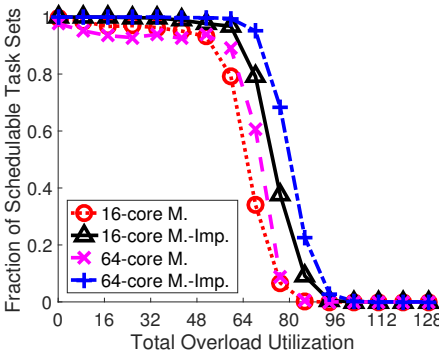
(d) For $U^N = 50\%m$.



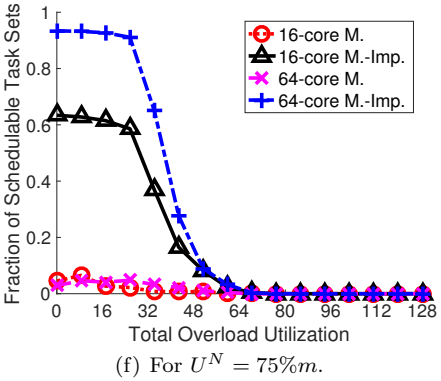
(b) For $U^N = 25\%m$.



(e) For $U^N = 62.5\%m$.



(c) For $U^N = 37.5\%m$.



(f) For $U^N = 75\%m$.

Fig. 8 Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on ***16 cores*** (labeled "16-core") vs. ***64 cores*** (labeled "64-core") with $\mathbf{p_{max}} = \frac{\mathbf{1.5}}{\mathbf{b}}$. From (a) to (f), figures show the results for varying total ***nominal utilizations***. Each figure shows the results for increasing total ***overload utilizations***.

10.2 Impact of varying U^N and U^O

We first evaluate the schedulabilities of MCFS-Bound MCFS and MCFS-Improve on a 32-core setting with $p_{max} = \frac{1.5}{b}$ in Figure 5 and 6. In particular, we select 6 representative total overload utilizations U^O ranging from $[12.5\%m, 25\%m, 37.5\%m, 50\%m, 62.5\%m, 75\%m]$, as shown in Figure 5(a) to 5(f). In each figure, we increase the total nominal utilization and plot the fraction of schedulable task sets of MCFS-Bound, MCFS and MCFS-Improve.

By the definition of capacity augmentation bound $2+\sqrt{2}$, the task sets that are deemed schedulable by ***MCFS-Bound*** are those with *both* L_i^N and L_i^O no more than $\frac{D_i}{2+\sqrt{2}}$ and with *both* U^N and U^O no more than $\frac{m}{2+\sqrt{2}} \approx 29\%m \approx 9.4$ on 32 cores. So only some of the task sets in the left one third of Figure 5(a), 5(b), 6(a) and 6(b) are schedulable by MCFS-Bound. Apparently, both MCFS and MCFS-Improve can admit many more task sets than is indicated by the capacity augmentation bound.

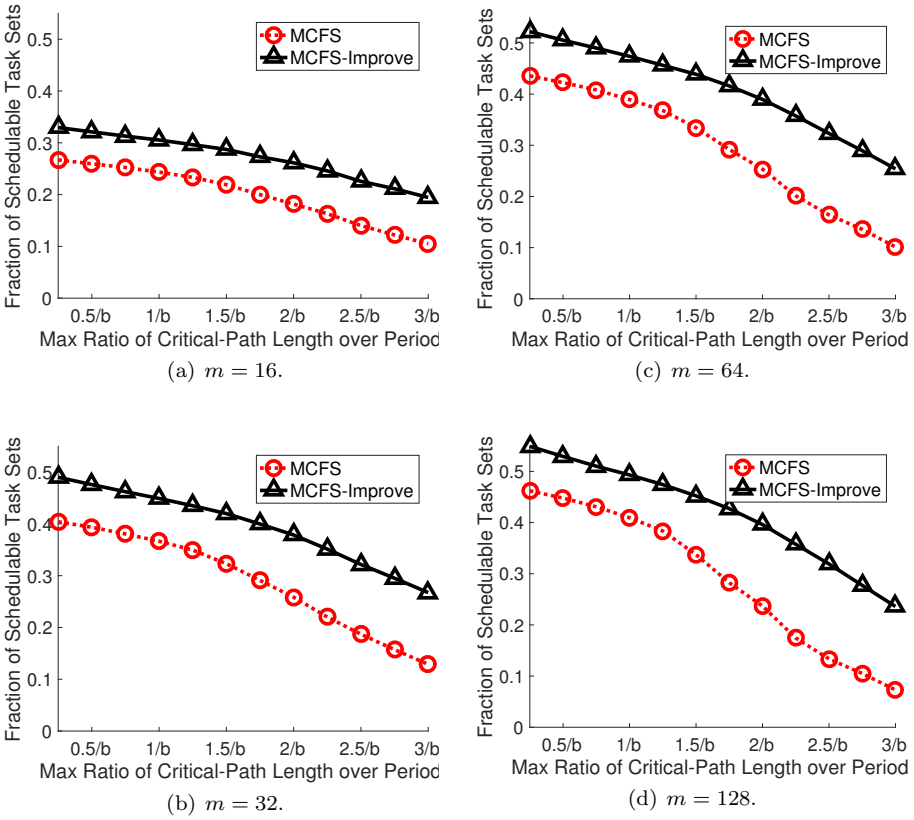
The trend of the Figure 5 and 6 shows that both MCFS and MCFS-Improve can schedule more task sets, when the total utilizations are lower. Moreover, we can observe that MCFS-Improve can schedule more task sets than MCFS, especially when the load of the system is high. For example, MCFS-Improve admits almost twice the number of task sets admitted by the basic MCFS schedulability test for $U^O = 75\%m$ in Figure 5(f) and 6(e); in Figure 6(f), almost no task sets are schedulable under basic MCFS, while MCFS-Improve can still schedule most task sets with overload utilization up to 8.

10.3 Impact of varying m

The trend of the schedulability comparison between MCFS and MCFS-Improve for 32 cores is similar to those of 16, 64 and 128 cores. In Figure 7 and 8, we can compare the fraction of schedulable task sets between the 16-core and 64-core setting on task sets with varying loads. We can observe that when the number of cores increases, both MCFS and MCFS-Improve can admit more task sets. Moreover, MCFS-Improve significantly improves over MCFS when the nominal utilization is high and the overload utilization is low in Figures 8(e) and 8(f). This is because in these cases there are many low-criticality tasks requiring many dedicated cores in the typical state. Hence, there are less cores remains for high-criticality tasks in typical state, while all cores are available for their low total overload utilizations in the critical state. For such cases, MCFS-Improve is able to find better virtual deadlines to decrease the number of cores assigned in the typical state and increase the number of cores assigned in the critical state, balancing the total core assignments in both states.

10.4 Impact of critical-path length

In Figure 9, we present the results with varying number of cores m and maximum ratio p_{max} of critical-path length over period. Each data point in each



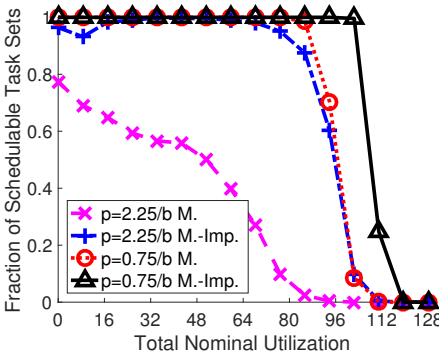
(a) $m = 16$. (c) $m = 64$.

(b) $m = 32$. (d) $m = 128$.

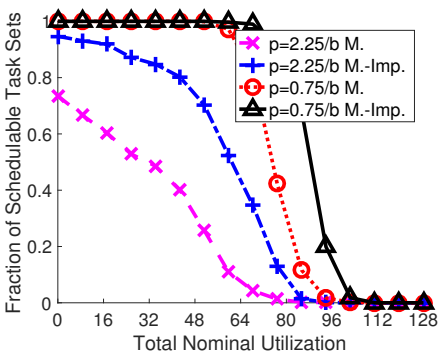
Fig. 9 Fraction of schedulable task sets of MCFS (dotted line) vs. MCFS-Improve (solid line) with varying number of cores. Each data point in each figure shows the average fraction of schedulable task sets of all the different settings (varying nominal and overload utilizations), given m and maximum ratio p_{max} of critical-path length over period.

figure shows the average fraction of schedulable task sets of all the settings with varying nominal and overload utilizations. For example, we have randomly generated and evaluated 256,000 task sets for the setting $m = 64$ and $p_{max} = \frac{2.5}{b} \approx 0.8$. MCFS can schedule 16.4% of these task sets, while MCFS-Improve can schedule 32.3% task sets. In this setting, MCFS-Improve has a relative improvement almost 100% over that of MCFS.

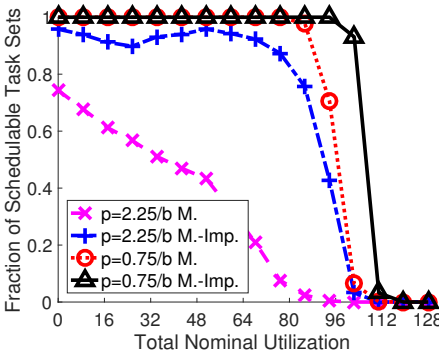
The ratio p_{max} has large impact on the schedulability of the family of MCFS algorithms, since it affects the parallelism of tasks. This is because given the same nominal and overload work, tasks with lower p_{max} have higher parallelism. In addition, when p_{max} decreases, tasks have more slack before the implicit deadline to complete their parallel work, so they require less dedicated cores. Therefore, MCFS and MCFS-Improve can schedule more task sets when p_{max} is lower. In addition, the improvement of MCFS-Improve increases with increasing p_{max}. This is because MCFS-Improve can adjust the virtual



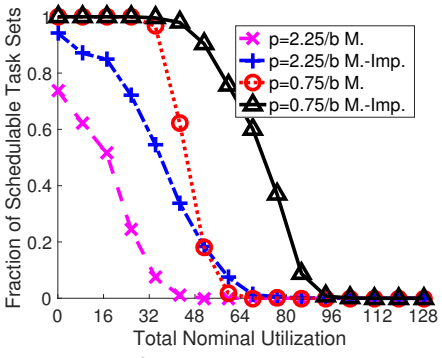
(a) For $U^O = 12.5\%m$ and $m = 128$.



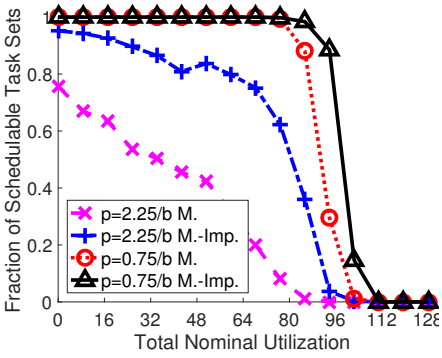
(d) For $U^O = 50\%m$ and $m = 128$.



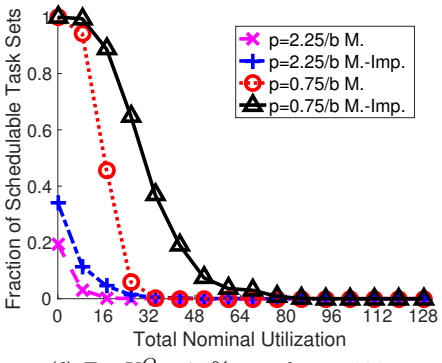
(b) For $U^O = 25\%m$ and $m = 128$.



(e) For $U^O = 62.5\%m$ and $m = 128$.

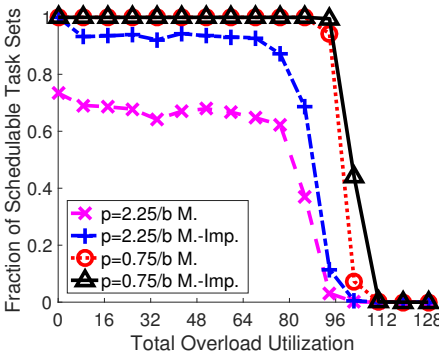


(c) For $U^O = 37.5\%m$ and $m = 128$.

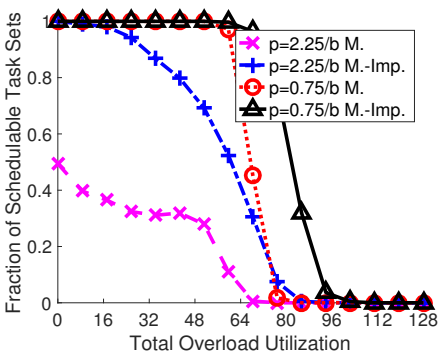


(f) For $U^O = 75\%m$ and $m = 128$.

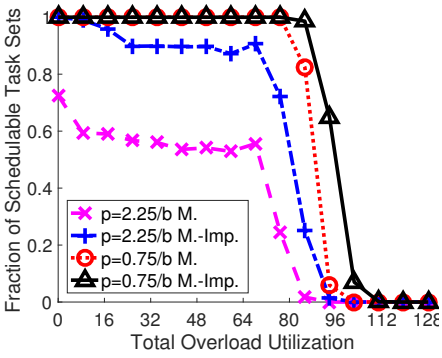
Fig. 10 Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on ***128 cores*** with $\mathbf{p_{max}} = \frac{\mathbf{0.75}}{\mathbf{b}}$ (labeled "p=0.75/b") vs. $\mathbf{p_{max}} = \frac{\mathbf{2.25}}{\mathbf{b}}$ (labeled "p=2.25/b"). From (a) to (f), figures show the results for varying total ***overload utilizations***. Each figure shows the results for increasing total ***nominal utilizations***.



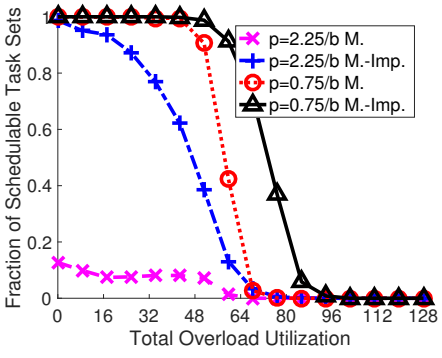
(a) For $U^N = 12.5\%m$ and $m = 128$.



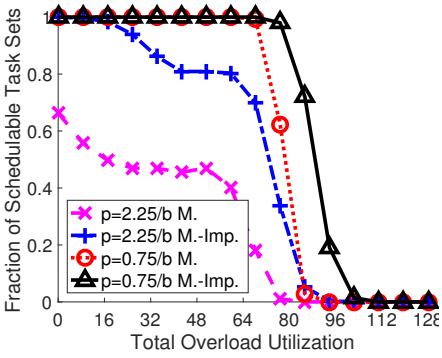
(d) For $U^N = 50\%m$ and $m = 128$.



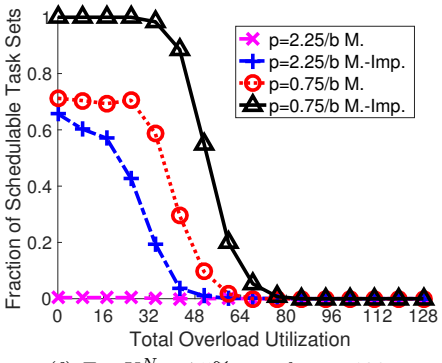
(b) For $U^N = 25\%m$ and $m = 128$.



(e) For $U^N = 62.5\%m$ and $m = 128$.



(c) For $U^N = 37.5\%m$ and $m = 128$.



(f) For $U^N = 75\%m$ and $m = 128$.

Fig. 11 Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on ***128 cores*** with $\mathbf{p_{max}} = \frac{\mathbf{0.75}}{\mathbf{b}}$ (labeled "p=0.75/b") vs. $\mathbf{p_{max}} = \frac{\mathbf{2.25}}{\mathbf{b}}$ (labeled "p=2.25/b"). From (a) to (f), figures show the results for varying total ***nominal utilizations***. Each figure shows the results for increasing total ***overload utilizations***.

deadline to balance the slacks before the virtual deadline and real deadline. This balances the core assignments in typical and critical states, so it results in considerably better schedulability.

To have a detailed look at the influences of the maximum critical-path length ratio p_{max}, we plot the results for 128-core with $p_{max} = \frac{0.75}{b}$ and $p_{max} = \frac{2.25}{b}$ in Figure 10 and Figure 11. Again, we can observe that MCFS and MCFS-Improve can admit more task sets when p_{max} is lower. However, p_{max} affect the performance of MCFS more than MCFS-Improve. By comparing these figures with previous figures, we can conclude that MCFS-Improve significantly improves over MCFS when tasks' critical-path lengths are relatively long, the number of cores is large, and/or the total nominal utilization is high.

11 Empirical Evaluation

We also evaluate our implementation of the MCFS runtime system described in Section 9 using synthetic workloads written in OpenMP. Experiments were conducted on a 16-core machine composed of two Intel Xeon E5-2687W processors (each with 8 cores). When running the experiments, we reserved two cores for operating system services, leaving 14 experimental cores. Linux with RT_PREEMPT patch version 4.1.7-rt8 was the underlying RTOS. For each setting, we randomly generate 100 task sets, each of which runs for 5 minutes — 300× the maximum period.

11.1 Task Set Generation

Now we explain how we generate task sets for the empirical evaluation. In these empirical experiments, the number of cores m is 14. We construct a task set by keep adding randomly generated tasks until MCFS schedulability test cannot admit any more tasks. Tasks are either high- or low-criticality with equal probability.

Note that the synthetic tasks in the empirical experiments are written in OpenMP. Each task has a sequence of parallel for loops, or segments. Each iteration of a segment is called a strand. We generate a task by first randomly choosing a desired overload critical-path length L', and then keep adding randomly generated segments until L' is reached.

The task parameters generation process is similar to (Saifullah et al, 2013). To generate tasks with large parallelism, we fix the maximum ratio p_{max} of the overload critical-path length over period as $p_{max} = \frac{1}{2(2+\sqrt{2})}$.

(1) Criticality z_i: 50% high-criticality and 50% low-criticality.
(2) Nominal and overload utilization ratio r_i for high-criticality tasks: uniformly from $[0.025, 0.25]$; The ratio r_i for low-criticality tasks is 1.
(3) Implicit deadline D_i: uniformly from 100ms to 1000ms.
(4) Max overload critical-path length L': 40%, 50%, 70% and 100% of $D_i p_{max}$, with probability of 0.4, 0.3, 0.2 and 0.1.

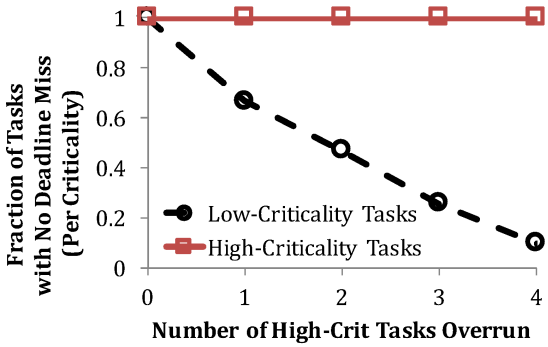


Fig. 12 Fraction of tasks with no deadline miss, for the sets of tasks with high- and low-criticality, respectively, when increasing the number of high-criticality tasks that overrun their nominal parameters.

(5) Number of strands of a segment $s_{i,j}$: randomly chosen from a log normal distribution with mean of $1 + \sqrt{m}/3$.
(6) Overload length of strands of a segment $t^O_{i,j}$: randomly chosen from a log normal distribution with mean of 5ms.
(7) Nominal length of strands of a segment $t^N_{i,j} = r_i t^O_{i,j}$.

With above parameters, we can calculate the nominal and overload work and critical-path length, which are used in MCFS schedulability test.

11.2 **Stress Testing**

We first conducted experiments to stress test the performance of the MCFS runtime system in both typical- and critical-states. In the typical-state stress testing, both high- and low-criticality task, execute exactly their worst-case *nominal* work and critical-path length. Experimental results are consistent with the correctness condition; no mode transition occurs and all high- and low-criticality tasks meet all their deadlines. In the critical-state stress testing, each task executes exactly its worst-case *overload* work and critical-path length. Again, in this worst case behavior, the result is also consistent with the correctness condition; every high-criticality task successfully transitions to critical-state and has no deadline miss. Some low-criticality tasks are preempted by high-criticality tasks, suspend some of their jobs and hence have deadline misses, which is allowed in critical-state.

11.3 **Graceful Degradation**

The mixed-criticality correctness condition allows us to discard all low-criticality tasks as soon as any task misses its virtual deadline and the system transitions to critical-state. However, the MCFS need not do so as discussed in Section 4.3. Figure 12 demonstrates that the MCFS runtime system can continue to run

many low-criticality tasks even after some high-criticality jobs transitions to critical-state. Here, we pick task sets with at least 4 high-criticality tasks. For each set, we run 5 experiments: either 0, 1, 2, 3 or 4 high-criticality tasks execute for their overload parameters and the remaining for their nominal parameters. We plot the fraction of tasks with no deadline miss. We can see that all high-criticality tasks always meet their deadlines. In contrast, the low-criticality task performance does not drop abruptly to zero as soon as the transition occurs. For instance, when only 1 high-criticality task overruns, only about 33% low-criticality tasks miss their deadlines.

11.4 **Recovery from Critical-State to Typical-State**

As discussed in Section 4, high-criticality tasks may optionally revert to the typical-state under the MCFS theory. Otherwise the transition to critical-state would be permanent and any low-criticality tasks sharing a processor would be permanently impaired. Here we construct a task system to illustrate the capability of the MCFS runtime system to recover from critical-state to typical-state, as shown below.

Table 7 Task Set Parameters

Task	C_i^N	C_i^O	D_i	D_i'	S^N	S^O
τ_1	5ms	20ms	20ms	10ms	1	1,2
τ_2	3ms	-	5ms	-	2	-

The task τ_1 is a high-criticality task constructed with the `rand()` function so that approximately 20% of the time it will require 20ms of computational effort, but the remainder of the time it will only require 5ms. Thus, roughly one-fifth of the time jobs of this task overruns its virtual deadline at 10ms, trigger a transition to critical-state, and involve the shared processor to help finish its computation on time. Task τ_1 is assigned with a single core (core 1) in the typical-state and it requires one additional core (core 2) in the critical-state. The computation (dense matrix multiplication) of the task is embarrassingly parallel, so, once activated, both threads will contribute nearly equal amounts of computational effort. Task τ_2 is a low-criticality task, which is assigned to core 2 in the typical-state. Hence, core 2 is shared among task τ_1 and τ_2.

The one hyper-period of the typical and critical situations are depicted in Figure 13 and Figure 14, respectively. If the system was not capable of recovering from the critical- to typical-state, task τ_1 would trigger a transition nearly immediately and task τ_2 would miss approximately half of its deadlines. However, with state recovery enabled, task τ_2 should miss two deadlines for each overrun of τ_1, but recover in those hyper-periods where τ_1 runs entirely in the typical-state. Indeed, when executed this is exactly what was found.

The task system was run for 100,000 hyper-periods. Task τ_1 missed zero deadlines but entered the critical state 21,087 times. Task τ_2 missed 42,174

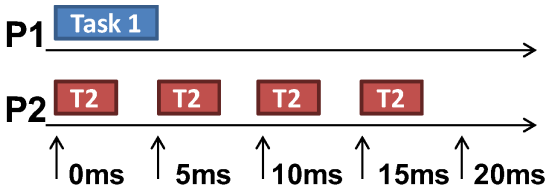


Fig. 13 One hyper-period of the experimental task system where Task 1 only requires 5ms of computational time.

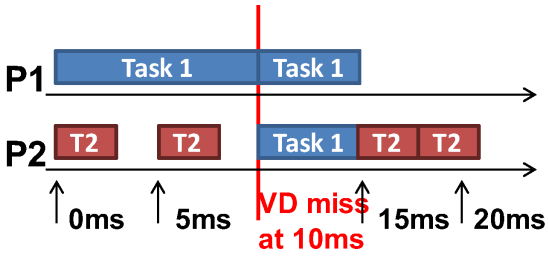


Fig. 14 One hyper-period of the experimental task system where Task 1 requires 20ms of computational time, necessitating a transition to critical-state.

deadlines out of 400,000, i.e., approximately 10.5% of the time. As confirmation, we also configured task τ_1 so that it always required 20ms of execution time and thus would always enter the critical-state. As expected, task τ_2 missed exactly half of its deadlines.

12 Related Work

In this section, we offer a survey of related work on real-time scheduling of both parallel tasks and mixed-criticality tasks.

Single-Criticality Scheduling on Multicores: There has been extensive research on scheduling single-criticality multiprocessor sequential tasks (Davis and Burns, 2011). Recently, many models of parallel tasks have been considered (Collette et al, 2008; Kato and Ishikawa, 2009; Lakshmanan et al, 2010; Lee and Heejo, 2006; Manimaran et al, 1998) and researchers have proved both resource augmentation bounds (Kim et al, 2013; Nelissen et al, 2012; Saifullah et al, 2013) and provided response time analyses (Andersson and de Niz, 2012; Chwa et al, 2013; Liu and Anderson, 2012) without decomposition. If we do not restrict tasks to parallel-for loops, we get the more general DAG model. For DAG tasks, G-EDF has a resource augmentation bound of $2 - \frac{1}{m}$ (Baruah et al, 2012b; Bonifaci et al, 2013; Li et al, 2013). Capacity augmentation bounds of 2 and 2.65 were proved for federated scheduling and G-EDF, respectively (Li et al, 2014).

Multi-Criticality Scheduling of Real-Time Tasks: Since (Vestal, 2007) first proposed a formal model for mixed-criticality systems, researchers have

studied scheduling sequential tasks on both single processor (Baruah et al, 2010, 2011; Easwaran, 2013; Ekberg and Yi, 2014; Gu et al, 2015; Guan et al, 2011; Lakshmanan et al, 2011; Liu et al, 2016; Socci et al, 2013) and multiprocessor machines (Pathan, 2012; Pellizzoni et al, 2009). In Section 2, we discussed the algorithms most relevant to our work that use virtual deadlines (Baruah et al, 2012a, 2014). Recently, MC-Fluid algorithm was proposed based on the fluid scheduling strategy (Baruah et al, 2015; Lee et al, 2014). This scheduler has been proved to have an optimal speedup bound of 4/3 for dual-criticality sequential tasks on multiprocessor systems. Models where other parameters, such as period and deadline, depend on the criticality of the task, have been investigated in (Baruah, 2012a, 2016a; de Niz and Phan, 2014; Su et al, 2014). See (Burns and Davis, 2016) for a comprehensive survey on mixed criticality scheduling.

None of these schedulers considers intra-task parallelism, however. (Baruah, 2012b) has considered limited forms of parallelism (such as that generated by Simulink programs), but the scheduling strategies are based on static scheduling tables which would be difficult to generate for more dynamic programs such as those from general purpose parallel programming languages such as OpenMP and Cilk Plus. Most recently, Liu et. al (Liu et al, 2014) consider scheduling of mixed-criticality synchronous tasks using a decomposition-based strategy. Like all decomposition strategies, the parallel task is decomposed into sequential tasks before runtime, so the task structure must be known in advance and cannot change between different jobs of the same task. Similarly, a mixed-criticality federated scheduling algorithm that considers more detailed parallel task information was studied in (Baruah, 2016b). In contrast, in this work we consider a more general DAG model and do not assume that the scheduler knows the structure of the DAG in advance, allowing the task to generate different DAG structures in each run.

13 Conclusions

In this paper, we presented the MCFS algorithm for dual- and multi-criticality parallel tasks. MCFS extends federated scheduling — a parallel real-time scheduling strategy for non mixed-criticality systems — to multi-criticality systems, by applying the virtual deadline technique proposed for sequential mixed-criticality tasks. We proved capacity augmentation bounds of MCFS for various conditions, which are the first augmentation bounds for parallel mixed-criticality systems. In addition, we provide the MCFS-Improve algorithm, that has better schedulability and should admit additional task sets. The numerical experiments show that the MCFS algorithm can schedule significantly more task sets than is indicated by the capacity bound. In addition, the MCFS-Improve algorithm has even better performance than MCFS, especially with higher load, more cores and tasks with longer critical-path length. To demonstrate the practicality of MCFS, we implemented a MCFS runtime system based on Linux and OpenMP, and conducted empirical evaluations.

There are several directions for future work. First, we want to explore other scheduling strategies which could potentially have better augmentation bounds. Second, we want to extend MCFS to schedule tasks with different task models, such as tasks with arbitrary deadlines and task activation like arrival curves. Finally, we assume that we have no knowledge about whether a job will present nominal or overload behavior. However, in practice, the system may have some indications as to whether a job will overrun or not. If we are provided with such information during runtime, how can we improve the scheduling decisions to meet more deadlines or to increase the system utilization?

References

Andersson B, de Niz D (2012) Analyzing global-edf for multiprocessor scheduling of parallel tasks. Principles of Distributed Systems pp 16–30

Baruah S (2012a) Certification-cognizant scheduling of tasks with pessimistic frequency specification. In: IEEE International Symposium on Industrial Embedded Systems (SIES), pp 31–38

Baruah S (2012b) Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In: 20th International Conference on Real-Time and Network Systems (RTNS), pp 11–19

Baruah S (2016a) The federated scheduling of systems of mixed-criticality sporadic dag tasks. In: IEEE Real-Time Systems Symposium (RTSS), pp 227–236

Baruah S (2016b) Schedulability analysis for a general model of mixed-criticality recurrent real-time tasks. In: IEEE Real-Time Systems Symposium (RTSS), pp 25–34

Baruah S, Li H, Stougie L (2010) Towards the design of certifiable mixed-criticality systems. In: 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 13–22

Baruah S, Burns A, Davis R (2011) Response-time analysis for mixed criticality systems. In: 32nd IEEE Real-Time Systems Symposium (RTSS), pp 34–43

Baruah S, Bonifaci V, D'Angelo G, Li H, Marchetti-Spaccamela A, Van Der Ster S, Stougie L (2012a) The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In: Real-Time Systems (ECRTS), 24th Euromicro Conference on, pp 145–154

Baruah S, Chattopadhyay B, Li H, Shin I (2014) Mixed-criticality scheduling on multiprocessors. Real-Time Systems 50(1):142–177

Baruah S, Eswaran A, Guo Z (2015) Mc-fluid: simplified and optimally quantified. In: IEEE Real-Time Systems Symposium (RTSS), pp 327–337

Baruah SK, Bonifaci V, Marchetti-Spaccamela A, Stougie L, Wiese A (2012b) A generalized parallel task model for recurrent real-time processes. In: 33rd IEEE Real-Time Systems Symposium (RTSS), pp 63–72

Bonifaci V, Marchetti-Spaccamela A, Stiller S, Wiese A (2013) Feasibility analysis in the sporadic dag task model. In: 25th Euromicro Conference on Real-Time Systems (ECRTS), pp 225–233

Burns A, Davis R (2016) Mixed criticality systems: A review. Department of Computer Science, University of York, Tech Rep

Chwa HS, Lee J, Phan KM, Easwaran A, Shin I (2013) Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In: Real-Time Systems (ECRTS), 25th Euromicro Conference on, pp 25–34

Collette S, Cucu L, Goossens J (2008) Integrating job parallelism in real-time scheduling theory. Information Processing Letters 106(5):180–187

Davis RI, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys 43(4):35

Dhall SK, Liu C (1978) On a real-time scheduling problem. Operations Research 26(1):127–140

Easwaran A (2013) Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In: 34th IEEE Real-Time Systems Symposium (RTSS), pp 78–87

Ekberg P, Yi W (2014) Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. Real-time systems 50(1):48–86

Ferry D, Bunting G, Maghareh A, Prakash A, Dyke S, Agrawal K, Gill C, Lu C (2014) Real-time system support for hybrid structural simulation. In: 14th International Conference on Embedded Software (EMSOFT), p 25

Gu X, Easwaran A, Phan KM, Shin I (2015) Resource efficient isolation mechanisms in mixed-criticality scheduling. In: Real-Time Systems (ECRTS), 27th Euromicro Conference on, pp 13–24

Guan N, Ekberg P, Stigge M, Yi W (2011) Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In: 32nd IEEE Real-Time Systems Symposium (RTSS), pp 13–23

Kato S, Ishikawa Y (2009) Gang edf scheduling of parallel task systems. In: 30th IEEE Real-Time Systems Symposium (RTSS), pp 459–468

Kim J, Kim H, Lakshmanan K, Rajkumar RR (2013) Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In: 4th International Conference on Cyber-Physical Systems (ICCPS), pp 31–40

Lakshmanan K, Kato S, Rajkumar R (2010) Scheduling parallel real-time tasks on multi-core processors. In: 31st IEEE Real-Time Systems Symposium (RTSS), pp 259–268

Lakshmanan K, de Niz D, Rajkumar R (2011) Mixed-criticality task synchronization in zero-slack scheduling. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 47–56

Lee J, Phan KM, Gu X, Lee J, Easwaran A, Shin I, Lee I (2014) Mc-fluid: Fluid model-based mixed-criticality scheduling on multiprocessors. In: IEEE Real-Time Systems Symposium (RTSS), pp 41–52

Lee WY, Heejo L (2006) Optimal scheduling for real-time parallel tasks. IEICE transactions on information and systems 89(6):1962–1966

Li J, Agrawal K, Lu C, Gill C (2013) Analysis of global edf for parallel tasks. In: Real-Time Systems (ECRTS), 25th Euromicro Conference on, pp 3–13

Li J, Chen JJ, Agrawal K, CLu, Gill C, Saifullah A (2014) Analysis of federated and global scheduling for parallel real-time tasks. In: Real-Time Systems (ECRTS), 26th Euromicro Conference on, pp 85–96

Li J, Ferry D, Ahuja S, Agrawal K, Gill C, Lu C (2016) A real-time scheduling service for parallel tasks. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 1–12

Liu C, Anderson J (2012) Supporting soft real-time parallel applications on multicore processors. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 18th International Conference on, pp 114–123

Liu D, Spasic J, Chen G, Guan N, Liu S, Stefanov T, Yi W (2016) Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In: IEEE Real-Time Systems Symposium (RTSS), pp 35–46

Liu G, Lu Y, Wang S, Gu Z (2014) Partitioned multiprocessor scheduling of mixed-criticality parallel jobs. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 20th International Conference on, pp 1–10

Manimaran G, Murthy CSR, Ramamritham K (1998) A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. Real-Time Systems 15(1):39–60

Nelissen G, Berten V, Goossens J, Milojevic D (2012) Techniques optimizing the number of processors to schedule multi-threaded tasks. In: 24th Euromicro Conference on Real-Time Systems (ECRTS), pp 321–330

de Niz D, Phan LT (2014) Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In: 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 111–122

OpenMP (2013) OpenMP Application Program Interface v4.0. http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Pathan RM (2012) Schedulability analysis of mixed-criticality systems on multiprocessors. In: Real-Time Systems (ECRTS), 24th Euromicro Conference on, pp 309–320

Pellizzoni R, Meredith P, Nam MY, Sun M, Caccamo M, Sha L (2009) Handling mixed-criticality in soc-based real-time embedded systems. In: 7th ACM international conference on Embedded software (EMSOFT), pp 235–244

Saifullah A, Li J, Agrawal K, Lu C, Gill C (2013) Multi-core real-time scheduling for generalized parallel task models. Real-Time Systems 49(4):404–435

Socci D, Poplavko P, Bensalem S, Bozga M (2013) Mixed critical earliest deadline first. In: Real-Time Systems (ECRTS), 25th Euromicro Conference on, pp 93–102

Su H, Guan N, Zhu D (2014) Service guarantee exploration for mixed-criticality systems. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 20th International Conference on, pp 1–10

Vestal S (2007) Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: 28th IEEE Real-Time Systems Symposium (RTSS), pp 239–243