Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks

Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, Chenyang Lu
Washington University in St. Louis
{li.jing,davidferry, shaurya.ahuja, kunal, cdgill, lu}@wustl.edu

***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. In addition, as the computational demands of real-time tasks increases, 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 real-time tasks modeled by directed acyclic graph. MCFS is based on federated intuition for scheduling parallel real-time tasks. It strategically assigns cores and virtual deadlines to tasks in order to achieve good schedulability. For task sets with only high-utilization tasks (utilization ≥ 1), we prove that MCFS provides a *capacity augmentation bound* of 3.41 and 3.73 for dual-criticality and multi-criticality, respectively. We also show that MCFS have capacity augmentation bounds of $3.67m/(m-1)$ for a dual-criticality system with both high- and low-utilization tasks, which to our knowledge is the first such performance bound for parallel mixed-criticality tasks. We also present an implementation of an MCFS runtime system in Linux that supports parallel programs written in OpenMP. We conduct both numerical and empirical experiments to demonstrate the practicality of our MCFS approach.**

I. 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 is 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.

Mixed-criticality model is an emerging paradigm for designing real-time systems, since 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.

In this paper, we study the problem of mixed-criticality scheduling of parallel real-time tasks. Parallel tasks are tasks with internal parallelism and can potentially use multiple cores. 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 [1] for a survey); and single-criticality scheduling of parallel tasks [2–12]. To our knowledge, there has been little prior work on the combined problem of mixed-criticality scheduling of parallel tasks, except for [13, 14].

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 [15], we generalize the idea of *capacity augmentation bounds* [11] 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.

978-1-4673-8641-8/16/$31.00 ©2016 IEEE

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 [12] to mixed-criticality systems. Federated scheduling (and thus MCFS) has the advantage of not requiring task decomposition. Likewise, the scheduler 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 structure of all the instances of the task. We assume that task sets are *implicit-deadline sporadic task sets*. This paper makes the following contributions.

Section III provides details of the MCFS algorithm and schedulability test for dual-criticality systems where all tasks are *high-utilization tasks* — 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 IV. To our knowledge, this is the first known augmentation bound for parallel mixed-criticality tasks.

In Section V, we generalize the MCFS algorithm and schedulability test for high-utilization task systems with more than two criticality levels. We also prove that the capacity augmentation bound in this case is $2+\sqrt{3} \approx 3.73$. Section VI 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).

We demonstrate the applicability of MCFS by implementing a MCFS runtime system in Linux that supports OpenMP parallel programs (Section VII). We conduct both numerical and empirical evaluations to show the practicality of MCFS (Section VIII). Empirical evaluation shows that the MCFS runtime system not only delivers mixed-criticality scheduling for parallel tasks, but also supports 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 will gradually discard low-criticality tasks as needed.

II. System Model and Background

Now we formally define the mixed-criticality parallel real-time task model and provide background from prior work.

A. 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 tast 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.

B. System Model for Dual-Criticality System

A dual-criticality system (we extend it to multi-criticality system in Section V) 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 typical-state, assuming each job will satisfy its nominal C_i^N and L_i^N. If, however, any job overruns its C_i^N or L_i^N, then the system *transitions* to the critical-state. Jobs of the low-criticality tasks then may be discarded, and 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 determining task set feasibility, the total utilization of a task set in 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 critical-state is the sum of the overload utilizations of high-criticality tasks $U^O = \sum_{\tau_i \in \tau \text{ and } Z_i = \text{HI}} u_i^O$.

C. 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.

D. Dual-Criticality Capacity Augmentation Bound

In [11] 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 $U^O = \sum_{\tau_i \in \tau \text{ and } Z_i = HI} C_i^O / D_i \leq m/b$.
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.

E. Most Relevant Prior Work

We now describe a couple of ideas that MCFS is based on.

Virtual Deadline: MCFS utilizes the idea of *virtual deadline*, first used in single processor mixed-criticality scheduler EDF-VD [16] and also used in multiprocessor mixed-criticality schedulers [17]. 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.

TABLE I. 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}

Federated Scheduling: MCFS is based on the federated scheduling approach [12] that assigns dedicated cores to high-utilization tasks and schedules them using a work-conserving scheduler. We use the following lemma, easily proved using Theorem 2 from [12], throughout this paper.

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

III. Scheduling Dual-Criticality High-Utilization Tasks

The MCFS scheduler consists of two parts: (1) a mapping algorithm (including schedulability test) runs before execution; and (2) a runtime scheduler. Before runtime, MCFS generates 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 either of 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 S^C.

In this section, we only consider dual-criticality systems where all tasks are high-utilization tasks.

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

In other words, a task is high-utilization, if it requires parallel execution to meet its deadline. Thus, MCFS allocates dedicated cores to all tasks. We consider task systems that contain both high and low-utilization tasks in Section VI. Table I shows the notation used throughout this paper.

A. 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 requirement in both states. To generate a mapping, MCFS classifies tasks into three categories.

TABLE II. TASK SET CLASSIFICATION AND ASSIGNMENT

Task Type	Task Type Classification: Criticality	Nominal Utilization	Overload Utilization	Assignment: 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	High	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$	$2+\sqrt{2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D'_i-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil\right\}$
HVH	High	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$	$2+\sqrt{2}$	$\frac{D_i}{b-1}$	1	$\left\lceil\frac{C_i^O-D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil$
LH	Low	$1 < u_i^N$	NA	2	D_i	$\left\lceil\frac{C_i^N-L_i^N}{D_i-L_i^N}\right\rceil$	NA

1) **Type 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) **Type 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.
3) **Type 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 II shows the classification criterion, 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.

B. Schedulability Conditions of MCFS

MCFS 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:

- 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 that it meets the correctness conditions from Definition 1. We will also show that MCFS provides a capacity augmentation bound of $2+\sqrt{2} \approx 3.41$.

C. 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 — tasks are allocated n_i^N dedicated cores and scheduled by a work-conserving scheduler.

If a high-criticality job J_i does not complete within its virtual deadline D'_i, 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.

Note on graceful degradation: After transitioning to the critical-state, MCFS *need 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 need 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 transition back to the typical-state by giving n_i^N cores to τ_i and re-admitting the low-criticality tasks that were suspended.

IV. PROOF OF CORRECTNESS AND CAPACITY AUGMENTATION BOUND

We now prove that for high-utilization tasks, as described in Section III, MCFS guarantees (1) correctness (as described in Definition 1) and; (2) a capacity augmentation bound of 3.41 (as described in Definition 2). We first prove properties of the mappings generated by MCFS for each of the three categories of tasks (from Table II). These properties then allow us to prove correctness and the 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 $\frac{a}{b} \geq \frac{a-cx}{b-x} \geq \frac{a-cy}{b-y}$;

A. 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 conditions. Since these tasks may be discarded in critical-state, we need only consider their typical-state behavior where they are assigned $n_i^N = \left\lceil\frac{C_i^N-L_i^N}{D_i-L_i^N}\right\rceil$ dedicated cores (see Table II) and $D'_i = D_i$ virtual deadline. 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 by $(b-1)$ times its utilization.

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 $b \geq 3$. ■

B. *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 nominal behavior and high utilization $u_i^O \geq 1$ in overload behavior. Table II shows that MCFS assigns each HVH task a single dedicated core in the typical-state ($n_i^N = 1$) and increases the number of allocated cores to $n_i^O = \left\lceil \frac{C_i^O - D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$ 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 sufficient to complete this work.

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 $n_i^O = \left\lceil \frac{C_i^O - D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil$ cores (by Ineq (1) and $\frac{C_i^O}{D_i} > 1$), which are sufficient by Lemma 1.

Case 2: The transition to critical-state occurred during the execution of 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 has finished executing already).

Let $e = t - r_i \leq D_i'$ be the amount of work that the job executed before the transition (since it has a dedicated core). Then, its remaining work is at most $C_i^O - e$, and its remaining critical-path length is at most L_i^O. The job has $D_i - e$ time to finish executing on n_i^O dedicated cores.

Since $u_i^O = C_i^O/D_i > 1$ and $e \leq D_i'$, by Ineq (1) we have $n_i^O = \left\lceil \frac{C_i^O - D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \geq \left\lceil \frac{(C_i^O - e) - L_i^O}{(D_i - e) - L_i^O} \right\rceil$. Hence, by Lemma 1, the allocated n_i^O cores are enough for it to be schedulable. ■

We now bound the number of cores assigned to HVH tasks in typical and critical-states. Since HVH tasks have high overload utilization $u_i^O \geq 1$, the following lemma is true.

Lemma 5. *For an HVH task τ_i, the number of assigned cores in the typical-state is $n_i^N = 1 \leq u_i^O$.*

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 $2 + \sqrt{2} \leq b \leq \frac{5+\sqrt{17}}{2}$.*

Proof: HVH task τ_i satisfies $u_i^O = C_i^O/D_i > 1$, and $L_i^O \leq D_i/b$. Therefore, by Ineq (1) we can derive

$$n_i^O < \frac{C_i^O - \frac{D_i}{b-1} - L_i^O}{D_i - \frac{D_i}{b-1} - L_i^O} + 1 \leq \frac{C_i^O - \frac{D_i}{b-1} - \frac{D_i}{b}}{D_i - \frac{D_i}{b-1} - \frac{D_i}{b}} + 1$$
$$= \frac{b(b-1)u_i^O - b - (b-1) + b^2 - 3b + 1}{b^2 - 3b + 1}$$

By solving the quadratic equation, we can conclude that for all b in range $[\frac{5-\sqrt{17}}{2}, \frac{5+\sqrt{17}}{2}]$, we have $-b - (b-1) + b^2 - 3b + 1 = b^2 - 5b + 2 \leq 0$. Therefore, we have $n_i^O \leq \frac{b(b-1)u_i^O}{b^2-3b+1}$.

Finally, $b \geq 2 + \sqrt{2}$, we have $b^2 - 3b + 1 \geq (b-1) > 0$. Thus, $n_i^O \leq \frac{b(b-1)u_i^O}{b-1} = bu_i^O$. By intersecting all the ranges of b, we get the required result $2 + \sqrt{2} \leq b \leq \frac{5+\sqrt{17}}{2}$. ■

Remarks: The classification and core assignment to HVH tasks may seem strange at first glance. Since the tasks have such low utilization in the typical-state, why do we assign dedicated cores rather than placing 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, Lemma 6 uses the fact that the task has a dedicated core 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.

C. *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 nominal behavior and high utilization $u_i^O > 1$ in overload behavior. Table II 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. *HVH tasks are schedulable under MCFS.*

Proof: In 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 work and L_i^N critical-path length within its virtual deadline $2D_i/b$. Thus, n_i^N cores are enough for it to be schedulable in 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 $n_i^O \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil$. 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 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 has 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 $C^O - n_i^N t^* - e + t^*$ 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:

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

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

(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 $n_i^O \geq n$, n_i^O cores are enough for HMH job j_i to be schedulable if the transition happens during its execution. ■

We now bound the number of cores assigned to HMH tasks in typical- and critical-states.

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 $n_i^N = \lceil u_i^O \rceil$.

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

Since $n_i^N \geq \lceil u_i^O \rceil \geq u_i^O$ and $D_i' = 2D_i/b$, we get $n' \leq \left\lceil \frac{C_i^O - u_i^O D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil = \left\lceil \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} \right\rceil$.

Since $C_i^O - \frac{2C_i^O}{b} = (1 - \frac{2}{b})u_i^O D_i > (1 - \frac{2}{b})D_i = D_i - \frac{2D_i}{b}$ and $L_i^O \leq D_i/b$, again applying Inequality (1) we can get

$$n' \leq \left\lceil \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} \right\rceil < \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} + 1 \leq \frac{C_i^O - 2C_i^O/b - D_i/b}{D_i - 2D_i/b - D_i/b} + 1 = \frac{(b-2)u_i^O - 1}{b-3} + 1$$

For all $4 > b > 3$, it is true that $1 - \frac{1}{b-3} < 0$. By solving the quadratic equation, we can conclude that for all $b \geq 2 + \sqrt{2}$, we have $b(b-3) - (b-2) = b^2 - 4b + 2 \geq 0$. Thus we get $n' < \frac{(b-2)u_i^O}{b-3} \leq \frac{b(b-3)u_i^O}{b-3} = bu_i^O$

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 for all $4 > b \geq 2 + \sqrt{2}$, $n_i^O = \max\{n_i^N, n'\} \leq bu_i^O$. ■

Remarks: 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.

D. *Proof of Correctness*

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

Theorem 10. *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.*

Remarks: Note that the correctness of MCFS does not rely on a particular b. In fact, the test can use any $b > 2$ to check the schedulability. Moreover, it can use different b's for different tasks. If the schedulability 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 use in our description are only in order to provide the capacity augmentation bound.

E. *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 utilization in both the typical and critical-state is no more than m/b and both critical-path lengths are no more than the deadline divided by b for all tasks, then MCFS will always declare the task set schedulable.

We first define some notations, summarized in Table I. 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 11. *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 (Section III-B): (1) valid virtual deadline — any task has $L_i^N \leq D_i'$ and any high-criticality task has $L_i^O \leq D_i - D_i'$; (2) valid typical-state mapping — $N_{LH}^N + N_{HVH}^N + N_{HMH}^N \leq m$; and (3) valid critical-state mapping — $N_{HVH}^O + N_{HMH}^O \leq m$.*

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

Lemma 12. *For any $b > 3$, if $L_i^N \le L_i^O \le D_i/b$ (Condition 3 of Definition 2), then the virtual deadline is always valid.*

Proof: **LHI tasks:** Virtual deadline $D_i' = D_i$, so $L_i^N \le D_i'$. **HVH tasks:** Virtual deadline $D_i' = D_i/(b-1) > D_i/b \ge L_i^N$ and $D_i - D_i' = (b-2)D_i/(b-1) > D_i/b \ge L_i^O$, since for any $b > 3$, we have $(b-2)/(b-1) > (b-2)/b \ge 1/b$. **HMH tasks:** Virtual deadline is $D_i' = 2D_i/b > D_i/b \ge L_i^N$ and $D_i - D_i' = (b-2)D_i/b > D_i/b \ge L_i^O$ for $b > 3$. ■

We now argue that a valid mapping S^T can be generated for typical-state for a capacity bound of $b = 3.41$.

Lemma 13. *If the Conditions of Definition 2 hold for any $b \ge 2 + \sqrt{2}$, then $m \ge N_{LH}^N + N_{HVH}^N + N_{HMH}^N$ and the typical state mapping S^T is valid.*

Proof: **LHI tasks:** From Lemma 3, for any $b \ge 2+\sqrt{2} > 3$, $N_{\text{LH}}^N = \sum_{\tau_i \in \tau_{\text{LH}}} n_i^N \le \sum_{\tau_i \in \tau_{\text{LH}}} (b-1)u_i^N = (b-1)U_{\text{LH}}^N$. **HVH tasks:** From Lemma 5 for any b, we always have $N_{\text{HVH}}^N = \sum_{\tau_i \in \tau_{\text{HVH}}} n_i^N \le \sum_{\tau_i \in \tau_{\text{HVH}}} u_i^O = U_{\text{HVH}}^O$. **HMH tasks:** From Lemma 8, for $b \ge 2 + \sqrt{2} > 2$, we have $N_{\text{HMH}}^N \le \sum_{\tau_i \in \tau_{\text{HMH}}} ((b-1)u_i^N + u_i^O) = (b-1)U_{\text{HMH}}^N + U_{\text{HMH}}^O$.

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

Therefore, we can derive

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

Thus, typical-state mapping is always valid if $b = 3.41$. ■

We now argue that the critical-state mapping S^C is valid for a capacity bound of $b = 3.41$.

Lemma 14. *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 $m \ge N_{HVH}^O + N_{HMH}^O$ and mapping S^C is valid.*

Proof: **HVH tasks:** From Lemma 6, for $b \ge 2 + \sqrt{2}$, we know $N_{\text{HVH}}^O = \sum_{\tau_i \in \tau_{\text{HVH}}} n_i^O \le \sum_{\tau_i \in \tau_{\text{HVH}}} bu_i^O = bU_{\text{HVH}}^O$. **HMH tasks:** From Lemma 9, for $b \ge 2 + \sqrt{2}$, we know $N_{\text{HMH}}^C = \sum_{\tau_i \in \tau_{\text{HMH}}} n_i^O \le \sum_{\tau_i \in \tau_{\text{HMH}}} bu_i^O = bU_{\text{HMH}}^O$.

Since the Conditions of Definition 2 hold, we also know $U^O = U_{\text{HVH}}^O + U_{\text{HMH}}^O \le m/b$.

Therefore, we get $N_{\text{HVH}}^C + N_{\text{HMH}}^C \le bU_{\text{HVH}}^O + bU_{\text{HMH}}^O \le m$

Thus, critical-state mapping is always valid if $b = 3.41$. ■

V. 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. Then, we will argue that MCFS provides the capacity augmentation bound of 3.73 for 3 or more criticality levels.

A. 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 system designer and overload work C_i^O estimated by certification authorities (similarly for critical-path length).[1] In all the criticality levels 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 lowest criticality level $Z_i = LO$: no action is taken if they overrun their nominal parameters. An example for three criticality levels $Z_i \in \{\text{LO}, \text{ME}, \text{HI}\}$ for low, medium and high, is shown in Table III.

TABLE III. MULTI-CRITICALITY TASKS PER-CRITICALITY WORK, CRITICAL-PATH LENGTH AND CORE ASSIGNMENT

	Work, Critical-Path Length			Core Assignment under MCFS		
Task Crit.	LO-Work	ME-Work	HI-Work	Typical-State	Intermed-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

A scheduler for a system with 3 criticality levels 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 column in Table III (i.e. nominal parameters for high-criticality tasks and overload parameters for medium-criticality tasks). The scheduler may discard all low-criticality tasks. (3) If any high-criticality task overrun its nominal parameters, then the system transition to the critical-state. High-criticality tasks still meet their deadlines based on their high-criticality parameters (i.e. overload parameters). The scheduler is allowed to discard all low and medium-criticality tasks.

B. Multi-Criticality MCFS Algorithm and Bound

We now generalize the MCFS algorithm to 3-criticality systems; further generalization is obvious and gives the same bounds. The classification and core assignments are shown in Table IV. Note that the classification is similar to the one shown in Section III. 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 in Table III. For instance, in the intermediate-state S^I, medium-criticality task gets n_i^O cores while high-criticality task gets n_i^N cores. In the schedulability test, we add an additional condition saying

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

TABLE IV. TASK SET CLASSIFICATION AND ASSIGNMENT FOR MEDIUM-CRITICALITY TASKS

	Task Type Classification			Assignment			
Task Type	Criticality	Nominal Utilization	Overload Utilization	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	High	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 \leq u_i^O$	$2+\sqrt{2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D_i'-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil\right\}$
HVH	High	$u_i^N \leq \frac{1}{b-1}$	$1 \leq u_i^O$	$2+\sqrt{2}$	$\frac{D_i}{b-1}$	1	$\left\lceil\frac{C_i^O-D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil$
MMH	Medium	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$	$\mathbf{2+\sqrt{3}}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D_i'-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil\right\}$
MVH	Medium	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$	$\mathbf{2+\sqrt{3}}$	$\frac{D_i}{b-1}$	1	$\left\lceil\frac{C_i^O-D_i'-L_i^O}{D_i-D_i'-L_i^O}\right\rceil$
LH	Low	$1 \leq u_i^N$	NA	2	D_i	$\left\lceil\frac{C_i^N-L_i^N}{D_i-L_i^N}\right\rceil$	NA

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 15. *Multi-criticality MCFS with only high-utilization tasks is correct and has a capacity bound of* $2+\sqrt{3}$.

Proof: Recall we noted in Section IV-D 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 $b = 2+\sqrt{3} \approx 3.73$, the correctness is obvious.

The proof of capacity augmentation is similar to Lemma 11 and we only point out the differences. We must bound the number of assigned cores in the intermediate-state mapping and prove that $N^O_{\text{MVH}}+N^O_{\text{MMH}}+N^N_{\text{HVH}}+N^N_{\text{HMH}} \leq m$. Note that in the this 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, we have $N^N_{\text{HVH}}+N^N_{\text{HMH}} \leq U^O_{\text{HVH}}+(b-1)U^N_{\text{HMH}}+U^O_{\text{HMH}}$, from Lemmas 5 and 8. Medium-criticality tasks are more interesting. In order to use a lemma similar to Lemma 13, we must bound n^O_{MVH} and n^O_{MMH}. Unfortunately, it is not sufficient to bound them by bU^O_{MVH} and bU^O_{MMH} as in Lemmas 6 and 9. Instead, we show a modified result similar to Lemma 6: $n_i^O \leq \frac{b(b-1)u_i^O}{b(b-1)-b-(b-1)} \leq (b-1)u_i^O$, holds for $4 > b \geq 2+\sqrt{3} \approx 3.73$. Similarly, a modified result for Lemma 9 requires $n_i^O \leq \frac{(b-2)u_i^O}{b-3} \leq (b-1)u_i^O$, which holds for $4 > b \geq (5+\sqrt{5})/2 \approx 3.62$. Thus, for $b \geq 2+\sqrt{3}$, we can show that $N^O_{\text{MVH}} \leq (b-1)U^O_{\text{MVH}}$ and $N^O_{\text{MMH}} \leq (b-1)U^O_{\text{MMH}}$. By taking summation and intersecting all the ranges of b, multi-criticality MCFS has a capacity bound of 3.73. ∎

Intuition: Why is the $2+\sqrt{2}$ for dual-criticality systems and $2+\sqrt{3}$ for all others? 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 the special property that lets us prove a better bound: when the system is in low-criticality mode, the low-criticality tasks also exhibit nominal behavior (instead of overload behavior).

VI. GENERAL CASE FOR DUAL-CRITICALITY MCFS

We can generalize MCFS to dual-criticality task systems with both high- and low-utilization tasks (both nominal and overload utilization is at most 1). The high-utilization tasks are still scheduled in a similar manner as in Section III 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 III, 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^N_{Seq} and U^O_{Seq}, 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 [17] to assign these tasks to cores. Say that the total numbers of assigned cores in typical- and critical-state to these tasks is N^N_{Seq} and N^O_{Seq}; 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^N_{Seq} 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 $N^O_{\text{Seq}} > N^N_{\text{Seq}}$, 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^O_{Seq} total cores for HLL tasks in overload behavior.

Correctness of this algorithm follows from Section IV-D and from the correctness of the chosen scheduler $\mathcal{S}$ for low-criticality tasks. The following theorem proves the capacity augmentation bound.

Theorem 16. *For dual-criticality systems with both high- and low-utilization tasks, MCFS has a capacity augmentation bound of* $(s+1)m/(m-1)$ *where* $s \geq 1+\sqrt{2}$ *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 [17], 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 III 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 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, respectively, are

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

As $b = (s+1)/(1-1/m)$, we can derive bound for typical-state mapping like Lemmas 14 and 13.

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

For critical-state mapping, we can also derive that $N^C_{\text{Seq}} + N^C_{\text{HVH}} + N^C_{\text{HMH}} \leq sU^O + 1 \leq sm/b + 1 \leq m$

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

VII. Implementation of a MCFS Runtime System

We demonstrate the applicability of MCFS, as described in Section III, by implementing a MCFS runtime system for a dual-criticality system with high-utilization tasks. This reference implementation supports parallel programs written in OpenMP [18]. It uses Linux with the 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 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^O_i). Each low criticality task creates n^N_i threads. (2) When the runtime system initializes (in typical-state), only n^N_i threads are awake for each task and they are pinned to distinct cores[4]. (3) The remaining $n^O_i - n^N_i$ 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.

Note that the threads of a task are 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 only imposed on low-criticality tasks by high-criticality tasks, so there is no criticality inversion.

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 $n^O_i - n^N_i$ threads, 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 default 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 $n^O_i - n^N_i$ 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. 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 occurs only once for each high-criticality task in the critical-state. To incorporate it into schedulability analysis, we subtract it from the deadline of each high-criticality task.

[2]As proved in [17], it has a speedup of $\frac{8}{3}$ compared to 100% utilization.

[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.

Impact of high-criticality tasks on low-criticality tasks: As discussed in Section VII, 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 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. This unexpected latency exposes an important limitation of this standard parallel concurrency platform when used in real-time systems.

```
//Called asynchronously by signal handler
barrier_state_switch()
    needs_switch = true

check_needs_updating()
    if( needs_switch )
        atomically_claim_switcher()
        if( switcher )
            verify_barrier_inactive()
            update_barrier_count()
            needs_switch = false
            release_spinwaiters()
        else spinwait()

mc_barrier_wait()
    check_needs_updating()
    do_barrier_wait()
```

Fig. 1. Mode Aware Barrier Psuedocode

State-Aware Barrier Implementation: One side-effect of the 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 of the threads in an OpenMP team are sleeping (as in our implementation), then 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` and implementing a state-aware barrier shown in Figure 1, 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 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.

VIII. Evaluation

A. Task Set Generation

First, we explain how we generate task sets for evaluation. For empirical and numerical experiments, tasks are generated in a similar manner, but task sets are constructed differently from tasks. In empirical experiments, there are $m = 14$ cores. We construct a task set by keep adding random tasks until MCFS schedulability test cannot admit any more tasks. Tasks are either high- or low-criticality with equal probability.

In contrast, for numerical experiments, each setting has desired total nominal and overload utilization U^N and U^O. We first add high-criticality tasks until U^O is reached and then add low-criticality tasks till 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" $r_{max} = U^N/U^O$ that cannot be exceeded by high-criticality tasks' nominal over overload utilizationss.

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 choose a desired overload critical-path length L', and then keep adding randomly generated segments until L' is reached.

To be consistent, we choose the similar approach to generate tasks for numerical experiments. However, it is important to note that for MCFS, the schedulability only depends on the work and critical-path length of tasks rather than their parallel structure (whether they are synchronous or DAG tasks).

Finally, we explain the task parameters generation process, which is similar to [3]. 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 task: for empirical tasks, uniformly from $[0.025, 0.25]$; for

numerical tasks, uniformly from 0.01 and a calculate r_{max}; This ratio r_i for low-criticality task is always 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.
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) Overload length of strands of a segment $t^O_{i,j} = r_i t^O_{i,j}$.

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

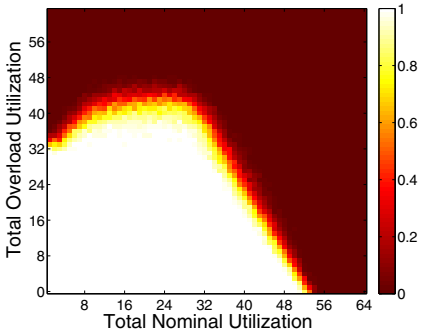

Fig. 2. Fraction of schedulable task sets by MCFS schedulability test with varying total nominal and overload utilizations on 64-core.

B. *Numerical Evaluation*

We first conduct a study comparing the schedulability test (Figure 2) and capacity augmentation bound of $2 + \sqrt{2}$ on $m = 64$ cores. Given an m core machine, we vary both the total nominal utilization U^N (x-axis) and overload utilization U^O (y-axis) of task sets from 1 to m. For each setting, we ran the MCFS schedulability test on 100 task sets and show the fraction of schedulable task sets admitted by MCFS. By definition, only the task sets with both U^N and U^O no more than $\frac{m}{2+\sqrt{2}} \approx 19$ are guaranteed to be schedulable by the MCFS capacity bound. Comparing it with Figure 2, we see the MCFS schedulability test outperforms the bound — MCFS test admits many more task sets with higher utilization. For example, MCFS test admits all task sets with $U^N = U^O = 30$.

Note that in Figure 2, when $32 < U^O < 48$, decreasing U^N actually makes it harder to schedule. This is because to fit in a small total U^N, the average nominal utilization of high-criticality tasks also becomes small. In other words, many of them are HVH tasks with very low nominal utilization, while assigned a single core by MCFS in typical-state. The number of assigned cores is hence many times more than their total utilization, making it harder to schedule.

C. *Empirical Evaluation*

We also evaluate our implementation of the MCFS runtime system described in Section VII 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 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 — $300\times$ the maximum period.

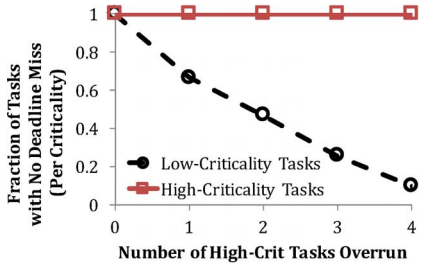

Fig. 3. Fraction of tasks with no deadline miss, for the sets of tasks with high- and low-criticality, respectively, when increasing the number of high-criticality tasks that overrun their nominal parameters.

Stress Testing: We first conducted experiments to stress test the performance of 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.

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, MCFS need not do so as discussed in Section III-C. Figure 3 demonstrates that MCFS runtime system can continue to run low-criticality tasks even after the transition. 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. Moreover, the low-criticality task performance does not drop abruptly to zero when the transition occurs. For instance, when 1 high-criticality task overruns, only about 33% low-criticality tasks miss their deadlines.

IX. Related Work

Now 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 [19]. Many models of parallel tasks have been considered [2, 20–23] and researchers have proved both resource augmentation bounds [3–5] and response time

analyses [6–8] without decomposition. For DAG tasks, G-EDF has resource augmentation bound of $2 - \frac{1}{m}$ [9–11]. Capacity augmentation bounds of 2 and 2.65 were proved for federated scheduling and G-EDF respectively [12].

Multi-Criticality Scheduling: Since [24] first proposed a formal model for mixed-criticality systems, researchers have studied scheduling sequential tasks on both single processor [25–32] and multiprocessor machines [33, 34]. In Section II, we discussed the algorithms most relevant to our work that use virtual deadlines [16, 17]. Models where other parameters, such as period and deadline, depend on the criticality of the task, have been investigated in [35–37].

None of these schedulers considers intra-task parallelism, however. Baruah [14] has considered limited forms of parallelism (such as that generated by Simulink programs). Most recently, Liu et. al [13] 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. In contrast, MCFS considers a more general DAG model and does not assume that the scheduler knows the structure of the DAG in advance.

X. 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. To demonstrate the practicality of MCFS, we implemented a MCFS runtime system based on Linux and OpenMP, and conducted numerical and empirical evaluations.

Acknowledgment

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

References

[1] A. Burns and R. Davis, "Mixed criticality systems: A review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[2] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010.

[3] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[4] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *ICCPS*, 2013.

[5] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS*, 2012.

[6] B. Andersson and D. de Niz, "Analyzing global-edf for multiprocessor scheduling of parallel tasks," *Principles of Distributed Systems*, pp. 16–30, 2012.

[7] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for synchronous parallel tasks on multicore platforms," in *ECRTS*, 2013.

[8] C. Liu and J. Anderson, "Supporting soft real-time parallel applications on multicore processors," in *RTCSA*, 2012.

[9] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[10] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *ECRTS*, 2013.

[11] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *ECRTS*, 2013.

[12] J. Li, J.-J. Chen, K. Agrawal, C.Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *ECRTS*, 2014.

[13] G. Liu, Y. Lu, S. Wang, and Z. Gu, "Partitioned multiprocessor scheduling of mixed-criticality parallel jobs," in *RTCSA*, 2014.

[14] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *RTNS*, 2012.

[15] S. K. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[16] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *ECRTS*, 2012.

[17] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[18] OpenMP, "OpenMP Application Program Interface v4.0," July 2013, http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[19] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, 2011.

[20] W. Y. Lee and L. Heejo, "Optimal scheduling for real-time parallel tasks," *IEICE transactions on information and systems*, vol. 89, no. 6, pp. 1962–1966, 2006.

[21] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, no. 5, pp. 180–187, 2008.

[22] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *RTSS*, 2009.

[23] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems," *Real-Time Systems*, vol. 15, no. 1, pp. 39–60, 1998.

[24] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007.

[25] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, 2010.

[26] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *RTSS*, 2011.

[27] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *RTSS*, 2011.

[28] A. Easwaran, "Demand-based scheduling of mixed-criticality sporadic tasks on one processor," in *RTSS*, 2013.

[29] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *RTAS*, 2011.

[30] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," in *ECRTS*, 2013.

[31] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-time systems*, vol. 50, no. 1, pp. 48–86, 2014.

[32] X. Gu, A. Easwaran, K.-M. Phan, and I. Shin, "Resource efficient isolation mechanisms in mixed-criticality scheduling," in *ECRTS*, 2015, pp. 13–24.

[33] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha, "Handling mixed-criticality in soc-based real-time embedded systems," in *EMSOFT*, 2009.

[34] R. M. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *ECRTS*, 2012.

[35] S. Baruah, "Certification-cognizant scheduling of tasks with pessimistic frequency specification," in *Industrial Embedded Systems (SIES), 7th IEEE International Symposium on*, 2012.

[36] H. Su, N. Guan, and D. Zhu, "Service guarantee exploration for mixed-criticality systems," in *RTCSA*, 2014, pp. 1–10.

[37] D. de Niz and L. T. Phan, "Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms," in *RTAS*, 2014.