Intractability Issues in Mixed-Criticality Scheduling

Kunal Agrawal[1]

Washington University in St. Louis,
St. Louis, MO, USA
kunal@wustl.edu
https://orcid.org/0000-0001-5882-6647

Sanjoy Baruah[2]

Washington University in St. Louis
St. Louis, MO, USA
baruah@wustl.edu
https://orcid.org/0000-0002-4541-3445

Abstract

In seeking to develop mixed-criticality scheduling algorithms, one encounters challenges arising from two sources. First, mixed-criticality scheduling is an inherently an on-line problem in that scheduling decisions must be made without access to all the information that is needed to make such decisions optimally – such information is only revealed over time. Second, many fundamental mixed-criticality schedulability analysis problems are computationally intractable – NP-hard in the strong sense – but we desire to solve these problems using algorithms with polynomial or pseudo-polynomial running time. While these two aspects of intractability are traditionally studied separately in the theoretical computer science literature, they have been considered in an integrated fashion in mixed-criticality scheduling theory. In this work we seek to separate out the effects of being inherently on-line, and being computationally intractable, on the overall intractability of mixed-criticality scheduling problems. Speedup factor is widely used as quantitative metric of the effectiveness of mixed-criticality scheduling algorithms; there has recently been a bit of a debate regarding the appropriateness of doing so. We provide here some additional perspective on this matter: we seek to better understand its appropriateness as well as its limitations in this regard by examining separately how the on-line nature of some mixed-criticality problems, and their computational complexity, contribute to the speedup factors of two widely-studied mixed-criticality scheduling algorithms.

1 Introduction

In the decade or so since it was first proposed as a formal model for representing mixed-criticality workloads, the Vestal model [15] has been the focus of a large body of scheduling-theoretic research (see [6] for a survey). One reasonable high-level "meta" conclusion that may be drawn from this research is that it is remarkably challenging to come up with general

algorithms for mixed-criticality scheduling that are efficient in terms of both running time and resource utilization; although impossibility and intractability results abound, relatively few efficient algorithms have been derived.

In this paper we report some of the findings of our ongoing efforts at obtaining a comprehensive understanding of the phenomenon of mixed-criticality scheduling, and seeking to explain why it is so difficult to schedule mixed-criticality systems efficiently. We separate out two distinct sources of intractability in mixed-criticality scheduling: (i) mixed-criticality scheduling is inherently an *on-line* problem, in which information needed to make good scheduling decisions is only revealed gradually during run-time; and (2) even ignoring this on-line nature, some basic mixed-criticality scheduling problems are *computationally intractable*. (For instance, it is known [5, Theorem 1] that determining whether a given collection of independent mixed-criticality jobs is schedulable is NP-hard in the strong sense.)

Now these two sources of intractability – being an on-line problem and being computationally intractable – have traditionally been considered separately in the theoretical computer science community:

1. The *competitive ratio/ factor* metric is used to quantify the sub-optimality of an on-line algorithm vis-à-vis an optimal clairvoyant one that is assumed to have complete knowledge about run-time behavior prior to making any scheduling decisions.

   For example, it is known that given a cache memory of k pages, the Least-Recently Used (LRU) paging algorithm is *k-competitive* [12] – upon some sequences of page requests it may experience up to k times as many page-faults as on optimal clairvoyant algorithm would.

2. In contrast, the *approximation ratio/ factor* metric quantifies the performance degradation resulting from using a polynomial-time algorithm for solving an NP-hard problem approximately.

   It is known, for example, that while the problem of scheduling a directed acyclic graph on a multiprocessor platform to minimize the makespan is NP-hard in the strong sense [14], List Scheduling [8] is a 2-approximation algorithm for this problem that runs in polynomial time: if a given directed acyclic graph can be scheduled optimally upon a specified multiprocessor platform to have makespan M, then List Scheduling will schedule it to have a makespan $< 2M$.

It is widely recognized in the theoretical computer science community that these two sources of intractability are fundamentally different from one another; however in the mixed-criticality scheduling theory literature a single metric – the speedup factor – tends to be used to quantify the effectiveness of mixed-criticality scheduling algorithms in dealing with both sources of intractability. As used in the mixed-criticality scheduling literature, the *speedup factor* metric of an algorithm A is the minimum multiplicative factor by which the speed of a computing platform must be increased in order that A be able to schedule any problem instance that is schedulable by some optimal clairvoyant scheduler. Used in this manner, it is particularly appropriate for quantifying the penalty arising from the on-line nature of mixed-criticality scheduling, but what about the penalty that may arise from computational intractability issues? This fundamental question motivated some lively discussions during a 2017 Dagstuhl Seminar on mixed-criticality systems,[3] regarding the benefits and drawbacks of using speedup factor as a quantitative metric to evaluate mixed-criticality scheduling algorithms. The opinion was expressed that since it compares algorithm A to an optimal *clairvoyant* scheduler,

---

[3] Dagstuhl Seminar 17131: *Mixed Criticality on Multicore/ Manycore Platforms*, March 26–31, 2017. `http://www.dagstuhl.de/17131`.

speedup factor is not very useful in comparing different on-line (non-clairvoyant) algorithms; a meaningful metric would not empower the adversary against whom each algorithm is being compared with as extreme a power as clairvoyance. The opposing opinion was that speedup factors as used in the mixed-criticality scheduling literature are merely an instantiation of the concept of the competitive factor metric [12], that has long been considered the gold standard for quantifying on-line algorithms – as a specific example, competitive factors continue to be widely used for justifying the choice of paging algorithms, despite the non-existence of clairvoyant paging algorithms. (Some other limitations of speedup factor as a metric for scheduling algorithms have been elaborated upon in [7]; rather than elaborate upon these limitations here we encourage the interested reader to peruse [7] since a discussion of these limitations is somewhat orthogonal to our prime objective here of niggling out the different effects of non-clairvoyance and computational complexity.)

This research. In this research, we attempt to obtain a better understanding of the role that speedup factor plays in characterizing mixed-criticality scheduling algorithms. We focus upon three widely-studied uniprocessor mixed-criticality scheduling problems that have previously been quantified with speedup factors: (i) scheduling of collections of independent dual-criticality *jobs*; (ii) scheduling of collections of independent dual-criticality *implicit-deadline periodic tasks*; and (iii) scheduling of collections of independent dual-criticality *implicit-deadline **sporadic** tasks*. For each problem, we consider three forms of schedulability:

1. CLAIRVOYANT SCHEDULABILITY: Given a dual-criticality instance I, can I be scheduled correctly[4] by a clairvoyant scheduling algorithm?
2. MC SCHEDULABILITY: Given a dual-criticality instance I, can I be scheduled correctly by an on-line (non-clairvoyant) scheduling algorithm?
3. A-SCHEDULABILITY (for some specified scheduling algorithm A): Given a dual-criticality instance I, can I be scheduled correctly by the scheduling algorithm A?
   In this work, the specific algorithms A that we consider are OCBP [4] for the job-scheduling problem, and EDF-VD [3] for the periodic and sporadic task-scheduling problems.

Computational complexity. The computational (in)tractability status of each of these schedulability problems is known to be as follows

- For each of the three problems that we study it is known (or can be easily deduced from prior results) that clairvoyant schedulability, as well as the respective A-schedulabilities (i.e., OCBP-schedulability for jobs; EDF-VD-schedulability for periodic and sporadic tasks) can be determined efficiently in polynomial time.
- What about MC-schedulability? The state of knowledge here is rather more sparse:
  - It has previously [5] been shown that determining MC-schedulability for a collection of independent dual-criticality jobs is NP-hard in the strong sense.
  - To our knowledge, no non-trivial prior results are known regarding MC-scedulability of dual-criticality periodic or sporadic implicit-deadline task systems. As one of the major contributions of this paper, we show that *determining MC-schedulability for collections of independent dual-criticality periodic and sporadic implicit-deadline tasks is also NP-hard in the strong sense.*

    The significance of this result for our purposes cannot be over-stated: as is the case for OCBP and the scheduling of independent jobs, it follows that EDF-VD, too, is dealing

---

[4] Precise definitions of what it means to schedule a dual-criticality system correctly are provided in Section 2.
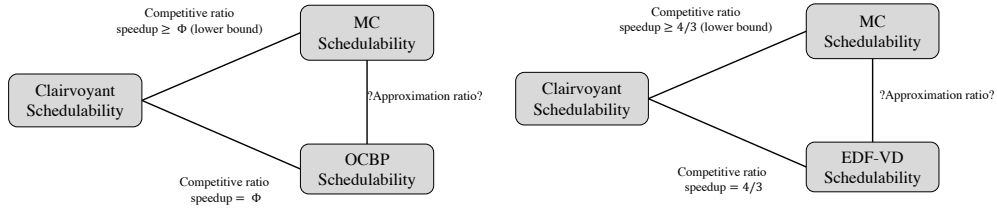
Figure 1 Summarizing the current state of our knowledge regarding speedup bounds for MC-scheduling of dual-criticality independent jobs (left figure) and dual-criticality implicit-deadline periodic & sporadic tasks (right figure). Each figure depicts previously-known lower and upper speedup bounds on the two sources of intractability: the on-line nature of MC-scheduling (given by speedup ratio), and the computational tractability of MC-schedulability analysis (given by approximation ratio).

with both the intractability arising from its non-clairvoyance and the intractability arising from having to solve an NP-complete problem in polynomial time. Hence its sub-optimality (as quantified by its speedup factor), too, can be attributed to two distinct sources.

Non-clairvoyance. The inherent intractability arising from the on-line nature of mixed-criticality scheduling problems has also been studied, and is fairly well understood for our three problems of interest:

- For collections of independent dual-criticality jobs, it was shown [4] that there are clairvoyant-schedulable instances that are not MC-schedulable without speedup $< \Phi$, where $\Phi = (\sqrt{5} + 1)/2 \approx 1.618$ denotes the Golden Ratio:
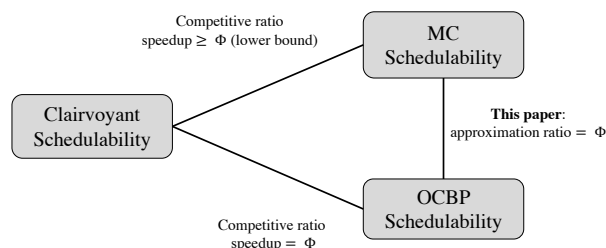
$$\Phi = \frac{\sqrt{5} + 1}{2} \quad \approx \quad 1.618 \tag{1}$$

- For collections of independent dual-criticality implicit-deadline sporadic tasks, it was shown [3] that there are clairvoyant-schedulable instances that are not MC-schedulable without speedup $< 4/3$.
- The proof in [3] is easily adapted from sporadic to periodic tasks, to show that there are clairvoyant-schedulable instances of independent dual-criticality implicit-deadline periodic tasks that are not MC-schedulable without speedup $< 4/3$.

The intractability results described above – computational intractability and loss of performance vis-à-vis clairvoyant schedulability – unfortunately do not fit in neatly with known results concerning specific scheduling algorithms. It is known, for instance, that the polynomial-time algorithm OCBP [4] is able to schedule any clairvoyant-schedulable collection of independent dual-criticality jobs with a speedup Φ. This implies that the speedup ratio Φ that is so widely used to characterize the effectiveness of OCBP as a mixed-criticality scheduling algorithm, is accounted for entirely by the fact that OCBP is solving a problem for which there is a lower bound of Φ on the speedup factor of any non-clairvoyant scheduling algorithm. This result is somewhat paradoxical: the approximation ratio of OCBP vis-à-vis MC-schedulability, arising from the fact that OCBP is only solving this problem approximately (which it inevitably is, since OCBP is a polynomial-time algorithm while determining MC-schedulability for collections of independent dual-criticality jobs is NP-hard in the strong sense) is not accounted for at all – see the left diagram in Figure 1. Similarly,

it is known that EDF-VD can schedule any clairvoyant-schedulable collection of independent dual-criticality implicit-deadline periodic or sporadic tasks with a speedup 4/3'rds – this is depicted in the right diagram in Figure 1; this again fails to account for the fact that EDF-VD is solving, in polynomial time, the MC-schedulablity problem for implicit-deadline periodic and sporadic tasks despite our showing, in this paper, that this problem is NP-hard in the strong sense.

One of our major results here is a proof that the *approximation ratio of OCBP* – i.e., its degradation in performance vis-à-vis MC-schedulability – is also quantified by a speedup factor equal to Φ. We show this by synthesizing an independent dual-criticality job instance that is MC-schedulable, but that can only be scheduled by OCBP upon a platform that is Φ times as fast. This immediately yields the interesting conclusion that the speedup factors quantifying (i) the intractability arising from the on-line nature of mixed-criticality scheduling , and (ii) the intractability arising from the computational complexity (NP-hardness) of recognizing MC-schedulability, do **not** "compose" in any meaningful sense: while each is equal to Φ in this case, the speedup factor of OCBP-schedulability vis-à-vis clairvoyant schedulability is also equal to Φ:

Organization. The remainder of this paper is organized as follows. In Section 2 we briefly describe the mixed-criticality workload models we will be using, and define some relevant concepts. In Section 3 we study the problem of scheduling collections of independent dual-criticality jobs. In Section 4 we present our results concerning the scheduling of collections of dual-criticality implicit-deadline recurrent (periodic and sporadic) task systems. We conclude in Section 5 with an enumeration of some interesting open issues and questions.

2 Model and Background

A mixed-criticality (MC) implicit-deadline recurrent (i.e., periodic or sporadic) task system τ consists of a finite specified collection of MC implicit-deadline recurrent tasks, each of which may generate an unbounded number of MC jobs.

MC jobs. As stated in Section 1 above, we will, for the most part, restrict our attention here to *dual*-criticality systems: systems with two distinct criticality levels. A *dual-criticality job* J_i is characterized by a tuple of parameters: $J_i = (\chi_i, a_i, [c_i^L, c_i^H], d_i)$, where

- $\chi_i \in \{L, H\}$ denotes the criticality of the job;
- $a_i \in R^+$ is the release time;
- c_i^L and c_i^H denote low-criticality and high-criticality estimates of the job's worst-case execution time (WCET) parameter; and
- $d_i \in R^+$ is the deadline.

System behavior.   The MC job model has the following semantics. Job J_i is released at time a_i, has a deadline at d_i, and needs to execute for some amount of time γ_i. The value of γ_i is not known beforehand, but only becomes revealed by actually executing the job until it *signals* that it has completed execution. These values of γ_i for a given run of the system defines the kind of *behavior* exhibited by the system during that run. If each J_i signals completion without exceeding c_i^L units of execution, we say that the system has exhibited LO-*criticality behavior*; if even one job J_i signals completion after executing for more than c_i^L but no more than c_i^H units of execution, we say that the system has exhibited HI-*criticality behavior*. If any job J_i does not signal completion despite having executed for c_i^H units, we say that the system has exhibited *erroneous behavior*.

Clairvoyance.   Before scheduling a collection of jobs, a *clairvoyant scheduling algorithm* knows, for each job J_i in the collection, the precise duration γ_i for which the job will need to execute prior to signaling completion. (Note that clairvoyant scheduling algorithms represent a hypothetical ideal that are not in general implementable in actual systems.). By contrast, an *on-line scheduling algorithm* does not know the values of γ_i beforehand; the value of γ_i is only revealed by executing J_i for a duration γ_i, at which instant it signals completion.

The notions of clairvoyant and on-line scheduling algorithms extend in the obvious manner to the scheduling of recurrent tasks (discussed next).

MC implicit-deadline recurrent tasks.   Analogously to traditional (non-MC) implicit-deadline recurrent tasks, an MC implicit-deadline recurrent (*periodic* or *sporadic*) task τ_k is characterized by a four-tuple $(\chi_k, C_k^L, C_k^H, T_k)$, with the following interpretation. Task τ_k generates an unbounded sequence of jobs, with successive jobs being released **exactly** T_k time units apart if the task is periodic, and **at least** T_k time units apart if it is sporadic. Each such job has a deadline that is T_k time units after its release. The criticality of each such job is χ_k, and it has LO-criticality and HI-criticality WCET's of C_k^L and C_k^H respectively; we assume that $C_k^L \leq C_k^H$ for all tasks τ_k.

An MC *implicit-deadline periodic/ sporadic task system* is specified by specifying a finite number of such periodic/ sporadic tasks. As with traditional (non-MC) systems, a MC sporadic task system can potentially generate infinitely many different MC instances (collections of jobs), each instance being obtained by taking the union of one sequence of jobs generated by each task.

Correctness criteria.   We define an MC scheduling algorithm to be *correct* if it is able to schedule any system such that

- During all LO-criticality behaviors of the system, all jobs receive enough execution between their release time and deadline to be able to signal completion; and

- During all HI-criticality behaviors of the system, all HI-criticality jobs receive enough execution between their release time and deadline to be able to signal completion.

As defined in Section 1, a dual-criticality instance is said to be *clairvoyant schedulable* if it can be scheduled correctly by some clairvoyant scheduling algorithm, *MC-schedulable* if it can be scheduled correctly by some (non-clairvoyant) on-line algorithm; and *A*-schedulable for some specified scheduling algorithm A if it can be scheduled correctly by the algorithm A.

3 Scheduling Mixed-Criticality Jobs

In this section we look at a very simple mixed-criticality scheduling problem: that of scheduling instances that are collections of independent dual-criticality jobs upon a single preemptive processor. To our knowledge, this problem was first studied in [4], where an algorithm called OCBP was proposed for solving it; OCBP were further studied in [5, 10]. Recall that an instance is defined to be *clairvoyant-schedulable* if it is scheduled correctly by an optimal clairvoyant algorithm; *MC-schedulable* if it is scheduled correctly by some (non-clairvoyant) on-line algorithm; and *OCBP-schedulable* if it is scheduled correctly by OCBP. The following results were proved in [5, 10] (again, Φ denotes the golden ratio).

R1 There are clairvoyant-schedulable instances that are not MC-schedulable with speedup $< \Phi$.

R2 Determining MC-schedulability is NP-hard in the strong sense (even if all jobs in the instance have equal release dates).

R3 If an instance is clairvoyant schedulable, then it is OCBP-schedulable with speedup Φ.

Results **R1** and **R2** above reveal that the "difficulty" in the problem being solved by OCBP arises from two sources: **R1** tells us that no on-line algorithm, regardless of its run-time computational complexity, can solve it optimally, while from **R2** it appears unlikely that the polynomial-time OCBP is able to even solve the on-line problem exactly. Result **R1** above tells us that no (non-clairvoyant) on-line algorithm can have a competitive factor smaller than Φ. Analogously to the competitive factor metric for quantifying on-line algorithms, the effectiveness of polynomial-time algorithms for solving NP-hard problems approximately is commonly quantified using the *approximation ratio* metric. The result **R3** is therefore somewhat paradoxical, and reveals one of the shortcomings of speedup factor as a metric for mixed-criticality scheduling algorithms: the fact that OCBP is only approximately solving an NP-hard problem is not revealed in its speedup factor (which takes on the optimal value of Φ).

In this section we consider OCBP from the perspectives of comparing its performance to both a clairvoyant algorithm (here we look at its competitive factor) and an optimal non-clairvoyant algorithm (here, we examine its approximation ratio). Both metrics provide different perspectives on its "distance" from optimal behavior – its competitive factor is a measure of its distance from optimality due to its non-clairvoyance (its not knowing the future) while its approximation ratio is due to its computational limitations – it is solving an NP-hard problem in polynomial time.

3.1 Current State of the Art

The OCBP algorithm is known to be speedup-optimal for scheduling dual-criticality collections of independent jobs. Given such a dual-criticality instance I, OCBP aims to derive offline (i.e., prior to run-time) a total priority ordering of the jobs of I such that scheduling the jobs according to this priority ordering guarantees a correct schedule, where *scheduling according to priority* means that at each moment in time the highest-priority available job is executed.

The priority list is constructed recursively using the approach commonly referred to in the real-time scheduling literature as the "Audsley approach" [1, 2]. OCBP first identifies a lowest priority job: Job J_i may be assigned lowest priority if

- it is a low-criticality job (i.e., $\chi_i = L$) and there is at least c_i^L time between its release time and its deadline available if every other job J_j has higher priority and is executed for C_j^L time units; **or**

    ▪   it is a high-criticality job (i.e., $\chi_i = H$) and there is at least c_i^H time between its release time and its deadline available if every other job J_j has higher priority and is executed for C_j^H time units.

The above procedure is repeated on the set of jobs excluding the lowest priority job, until all jobs are ordered, or at some iteration no lowest priority job is identified (if this happens OCBP declares failure and exits: it is unable to schedule this instance).

    The results **R1**–**R3** listed above permit us to draw the following conclusions about the effectiveness of OCBP:

1. From **R1** and **R3**, we conclude that *OCBP has the optimal **competitive ratio** from the perspective of speedup-versus-clairvoyance.*

2. But that leaves unanswered the question of how far OCBP is from **on-line optimality** (as indicated by MC-schedulability). In contrast to **R2**, OCBP-schedulability can be determined in polynomial time – what is the **approximation ratio** of OCBP in comparison to an optimal on-line scheduler?

3.2   Additional Insights

We now describe some of our new findings that allow us to better characterize the effectiveness with which the polynomial-time OCBP algorithm approximates solutions to the NP-hard problem of MC-schedulability. We first show, in Section 3.2.1, that OCBP is in fact optimal for scheduling MC-schedulable instances comprising just two jobs. (Although this result may at first seem trivial, we point out that the proof of result **R1** was obtained using 2-job instances; hence the result proving optimality of OCBP vis-à-vis MC-schedulability serves to separate the intractability arising from non-clairvoyance from the intractability arising from computational intractability issues). This positive result contrasts with a negative result in Section 3.2.2, where we show that in general, however, OCBP's approximation ratio is at least Φ: there exist MC-schedulable instances that are not OCBP-schedulable with speedup $< \Phi$.

3.2.1   Instances with at most two jobs

We now show that OCBP is able to optimally schedule any instance comprising at most two jobs. The cases when the instance comprises zero or one jobs is trivial; let us therefore focus on two-job instances. Consider first the case when both jobs have the same release date (without loss of generality, assumed equal to zero). If both jobs have the same criticality, or if the high-criticality job has an earlier deadline, OCBP will find the EDF-schedule for the jobs, and this is the optimal schedule. Consider, therefore, the case when the earlier-deadline job has lower criticality. Consider the following instance:

$$\{J_1 = (L, 0, [c_1^L, -], d_1), J_2 = (H, 0, [c_2^L, c_2^H], d_2)\}$$

with $d_1 < d_2$.

    We claim that this instance is MC-schedulable if and only if (in addition to each job being "well-formed" – i.e., $c_i^L \le c_i^H$ – and individually feasible) the following condition is satisfied:

$$\left(\left(c_1^L + c_2^L \le d_1 \right) \bigvee \left(c_1^L + c_2^H \le d_2 \right) \right) \tag{2}$$

The reason why this is a necessary and sufficient condition for MC-schedulability is as follows:

◾ **Table 1** Example job instance depicting the non-optimality of OCBP vis-à-vis MC-schedulability. (Here ϵ denotes a positive constant < 1, and $y > 1$.)

	criticality	release date	wcets	deadline
J_1	high	0	$[\epsilon, 1]$	1
J_2	low	0	$[1 - \epsilon, 1 - \epsilon]$	1
J_3	high	0	$[y - 1, y - 1]$	y

◿ If the first disjunct is satisfied, we start out executing J_2.

This is clearly a correct scheduling strategy, since in any LO-criticality behavior both jobs will complete before the earlier deadline d_1 while in any HI-criticality behavior the HI-criticality job gets to execute to completion and hence (since each job is assumed to be individually feasible) meets its deadline.

◿ If the second disjunct is satisfied, we start out executing J_1.

This, too, is clearly correct: the LO-criticality job executes first and therefore (since it is assumed individually feasible) receives up to c_1^L units of execution by its deadline. It does not receive more than c_1^L units of execution regardless of whether it signals completion or not; hence, the disjunct ensures that the HI-criticality job always gets up to $c_@^H$ units of execution prior to its deadline.

◿ If neither disjunct is satisfied, then consider any schedule over the interval $[0, d_1]$:

  ▪ If job J_1 has executed for $< c_1^L$ units, then the instance reveals low-criticality behavior
  ▪ Else job J_1 has executed for c_1^L units (in which case job J_2 could not have executed for c_2^L units to reveal high-criticality behavior), and the instance reveals high-criticality behavior.

Now observe that if Condition 2 is satisfied, then OCBP would successfully schedule the system – if the first disjunct is true then J_1 can be assigned lower priority while if the second disjunct is true then J_2 could be assigned lower priority, by OCBP.

Now to generalize to the case where the two jobs may not have equal release dates, observe that any work-conserving algorithm would schedule the job that is released throughout the interval between its release and the release date of the second job (or to completion – whichever occurs first). And once the second job arrives, we are back to the case of two jobs with equal release dates, with the WCETs of the job that was released first appropriately modified to reflect the execution that has already occurred.

3.2.2 General Instances

We have seen that OCBP is optimal vis-à-vis MC-schedulability for dual-criticality instances with two or fewer jobs. Unfortunately, this optimality property does not hold for > 2 jobs; we show this by constructing a 3-job instance shown in Table 1. It is easy to show that *this instance is MC-schedulable*: an optimal on-line algorithm would execute J_1 to completion. If this occurs within ϵ time-units, then the optimal algorithm executes J_2 and then J_3; if not, it simply discards J_2 and executes J_3 to completion.

However, *this instance is not OCBP-schedulable*; we prove this by showing that none of the three jobs can be assigned lowest priority by OCBP (and hence OCBP would report failure at the very beginning – it is unable to assign any job lowest priority):

1. For J_1 to be lowest-priority, we would need

$$\Big(1 + (1 - \epsilon) + (y - 1) \leq 1\Big) \Leftrightarrow \Big(1 + y - \epsilon \leq 1\Big)$$

which is impossible since $\epsilon < 1$ and $y > 1$.

2. For J_2 to be lowest-priority, we would need

$$\Big(\epsilon + (1-\epsilon) + (y-1) \le 1\Big) \Leftrightarrow \Big(y \le 1\Big)$$

which is impossible since $y > 1$.

3. For J_3 to be lowest-priority, we would need

$$\Big(1 + (1-\epsilon) + (y-1) \le y\Big) \Leftrightarrow \Big(1 - \epsilon + y \le y\Big)$$

which is impossible since $\epsilon < 1$.

We saw above that the 3-job instance depicted in Table 1 is not OCBP-schedulable; we now compute the minimum speedup needed in order that this instance be OCBP-schedulable upon the faster platform. Since the instance is MC-schedulable this speedup value would represent a lower bound on the approximation ratio of OCBP (vis-à-vis MC-schedulability).

Note that OCBP must assign some job lowest priority. We consider each of the three jobs as potential candidate lowest-priority jobs:

1. For J_1 to get lowest priority, the processor would to complete $(1 + y - \epsilon)$ units of work by time-instant 1, in order that J_1 complete by its deadline at time-instant 1. The needed speedup is therefore

$$(1 + y - \epsilon) \tag{3}$$

2. For J_2 to get lowest priority, the processor would to complete y units of work by time-instant 1, in order that J_2 complete by its deadline, also at time-instant 1. The needed speedup is therefore

$$y \tag{4}$$

3. For J_3 to get lowest priority, the processor would to complete $(1 + y - \epsilon)$ units of work by time-instant y, in order that J_3 complete by its deadline at time-instant y. The needed speedup is therefore

$$\frac{1 + y - \epsilon}{y} = 1 + \frac{1 - \epsilon}{y} \tag{5}$$

Since $y > \epsilon$, Expression (4) < Expression (3) and hence J_2 requires a lower speedup than J_1 to be a viable lowest-priority job. Which of J_2 or J_3 needs a lower speedup to be a viable lowest-priority job depends upon the exact values of ϵ and y. Since the speedup needed for J_2 to be a viable lowest-priority job increases, while the speedup needed for J_3 to be a viable lowest-priority job decreases, with increasing y, the largest speedup needed occurs when the two values are equal:

$$y = 1 + \frac{1 - \epsilon}{y}$$
$$\Leftrightarrow \quad y^2 - y - (1 - \epsilon) = 0$$

As $\epsilon \to 0$, the solution to the quadratic equation above approaches the golden ratio $\Phi = (\sqrt{5} + 1)/2$, and the speedup needed is equal to this value of y.

We have therefore just shown that this is at least one MC-schedulable instance – the one depicted in Table 1 – for which OCBP needs a speedup of Φ. It has previously been shown that any clairvoyant-schedulable instance is OCBP schedulable with speedup Φ (this is the result **R3** referenced above), from which it follows that any MC-schedulable instance is also OCBP schedulable with speedup at most Φ. We therefore conclude that **the approximation ratio of OCBP is equal to** Φ, yielding the situation depicted in Section 1.

4 Sporadic Task Schedulability

We focus our attention in this section on dual-criticality implicit-deadline recurrent – sporadic and periodic – task systems. Our main contribution here is a proof that the mixed-criticality scheduling problem is NP-hard for such systems; this stands in sharp contrast to the analogous problem for single-criticality systems, where a simple linear-time utilization-based schedulability test is known [11]. This computational complexity result serves to establish that determining MC-schedulability is computationally intractable for dual-criticality recurrent task systems, and hence demonstrates that EDF-VD (like OCBP) is dealing with two sources of intractability: non-clairvoyance and computational complexity. This opens up the need for separating out the effects of the two sources of intractability, and also highlights the need for metrics that are able to separately quantify the approximation-ratio effect (i.e., comparison with MC-optimality) and the competitive-ratio effect (i.e., the sub-optimality arising from the lack of clairvoyance) of EDF-VD in scheduling recurrent sysems..

4.1 Task Set Construction

Recall that, dual criticality mixed criticality job scheduling is NP-complete in the strong sense [5, Theorem 1]. We will use that proof as a starting point for our proof to show how a similar reduction can also be used for both periodic and sporadic tasks. The reduction in [5, Theorem 1] is from an instance I of 3-PARTITION. The 3-PARTITION problem is the following: we are given a set S of $3m$ positive integers $s_0, s_1, s_2, ..., s_{3m-1}$ and a positive integer B such that $B/4 < s_i < B$ for each i, and $\sum_{i=0}^{3m-1} s_i = mB$. Instance I is said to have a feasible solution if S can be partitioned into m disjoint sets, $S_0, S_1, ..., S_{m-1}$ each of which sum to B.

Hardness construction for mixed-criticality jobs from [5, Theorem 1]. In [5, Theorem 1], the authors showed that dual-criticality job schedulability is solvable in polynomial time iff 3-PARTITION is solvable in polynomial time. Given an instance I of 3-PARTITION, they defined a polynomial-time procedure for generating a set X_I of dual-criticality jobs such that X_I can be scheduled correctly by an online scheduler iff I has a feasible solution.[5]

From I, the dual-criticality jobs set X_I is generated as follows:

- $3m$ HI-criticality jobs ($\chi_i = H$) each with release time 0, deadline $2mB$, $C_i^L = s_i$ and $C_i^H = 2s_i$.
- m LO-criticality jobs ($\chi_i = L$) each with release time iB for $0 \le i \le (m-1)$ and deadline $2B$ after its release and $C_i^L = C_i^H = B$.

The derivation in [5, Theorem 1] shows that this job system X_I is schedulable in an MC-correct manner iff I has a valid 3-partition. We will not recall the details of the proof that shows that if X_i is schedulable, then I is feasible since those details are not important for our proof here. We will briefly recall the direction that shows that if I is feasible then X_I is schedulable. To do so, given a feasible 3-PARTITION for I, we must provide a schedule for X_I such that all jobs are scheduled correctly.

Given a feasible partition $S_0, S_1, S_2, ..., S_{m-1}$, the feasible scheduling policy is as follows. Look at m chunks of time, each of size (i.e., duration) $2B$. In the first B time of chunk j, we execute the tasks in S_j – since the LO-criticality execution time of all jobs in S_j sums to B,

---

[5] To prove NP-completeness, one must also reduce from a set of jobs to an instance of 3-PARTITION; however, we are only concerned with NP-hardness in this paper.

this interval is sufficient to finish these jobs if the system stays in LO-criticality mode. In the second chuck of time, we execute the LO-criticality job. Therefore, all jobs are schedulable in LO-criticality behavior.

We now argue about HI-criticality behavior. Consider a job τ_i in set S_j. Say this job exceeds its LO-criticality execution time. Then the system will discover this by time $(2j+1)B$ since all jobs of set S_j will have completed their LO-criticality execution by then. At this point, the system will transition to HI-criticality mode and all LO-criticality jobs will be discarded. Therefore, the schedule has $(2m - 2j - 1)B$ time to finish the HI-criticality jobs' remaining computation before their deadline. At this point, all jobs in sets $S_0, S_1, ..., S_{j-1}$ have already signalled completion and the jobs in set S_j have completed B units of work. Therefore, even if all jobs in sets $S_{j+1}, ..., S_{m-1}$ execute for their HI-criticality execution time of $2s_i$, we have total of $(2m - 2j - 1)B$ total remaining HI-criticality work which can be completed by the deadline.

Translating the construction to dual-criticality tasks.   The construction in [5, Theorem 1] described above applies to dual-criticality jobs; now we will show how we can use a similar construction for dual-criticality tasks. Given a feasible instance of 3-PARTITION I, we construct a mixed-criticality job system Y_I as follows:

- $3m$ HI-criticality tasks ($\chi_i = H$) each with period $2mB$, $C_i^L = s_i$ and $C_i^H = 2s_i$.
- 1 LO-criticality task ($\chi_i = L$) with period $2B$ and $C_i^L = C_i^H = B$.

Note that if the system is periodic with the first release of all jobs at time 0, then Y_I generates an instance of dual-criticality jobs X_I in each hyper-period of $2mB$. Therefore, the following lemma about periodic jobs is obvious.

▶ **Lemma 1.** *If tasks are periodic, then this task set Y_I is schedulable iff the job system X_I was schedulable. Therefore, determining schedulability of periodic implicit deadline dual-criticality task systems is NP-hard in the strong sense.*[6]

It is not so straightforward, however, to argue that sporadic schedulability is NP-hard in the strong sense. We will now prove the following theorem in the rest of this section.

▶ **Theorem 2.** *A sporadic task system generated by Y_I is schedulable iff I has a feasible solution, that is, if the corresponding periodic task system is schedulable. Therefore, determining schedulability of sporadic implicit deadline dual-criticality task systems is NP-hard in the strong sense.*

It is clear that if the sporadic task system is schedulable, then the periodic one is also schedulable since periodic releases are just an instance of sporadic releases. The rest of this section will show that if the periodic task system is schedulable, then the sporadic one is also schedulable.

Note that we are not making a general claim about all task systems. It is sufficient to show the following: If an instance I of 3-PARTITION (set S of $3m$ positive integers $s_0, s_1, s_2, ..., s_{3m-1}$ and a positive integer B such that $B/4 < s_i < B$) is feasible (we can find m sets $S_0, S_1, ..., S_{m-1}$ where each set sums to B), then the corresponding dual-criticality sporadic task system Y_I is schedulable. We will do so by designing a particular scheduler for this type of task set.

---

[6] Again, we are concerned only with NP-hardness; therefore, we needn't show a reduction from periodic task instances to 3-PARTITION instances.

4.2   Scheduling a Sporadic Instance of Y_I

Here, we wish to design a schedule that can correctly schedule the sporadic task set if the periodic task set is schedulable. In other words, given a feasible instance of I, this scheduler will always meet all deadlines if no job exceeds its LO-criticality WCET and will meet all HI-criticality deadlines if some job exceeds its LO-criticality WCET, but does not exceed its HI-criticality WCET. Intuitively, this scheduler tries to mimic the periodic schedule.

LO-criticality mode scheduling algorithm.   The run-time scheduling algorithm is an almost-fixed-priority scheduler when in LO-criticality mode. In particular, each high-criticality task is assigned a priority and higher-priority high-criticality tasks take precedence over lower-priority high-criticality tasks. Priority assignment to the $3m$ HI-criticality tasks is determined according to the set they are in for the solution of I. Tasks in set S_0 have priority 0, tasks in S_1 have priority 1 and so on – here, lower numbers denote greater scheduling priority.

However, scheduling decisions for low-criticality jobs are a little different. Each HI-criticality task τ_i in set S_j maintains an auxiliary variable called *slack* $\ell_i \leftarrow jB$ – this is the amount of time the LO-criticality task is allowed to execute after τ_i is released before τ_i becomes "higher priority" than the LO-criticality task.

The schedule is a work-conserving schedule with the following properties. A HI-criticality job always yields to all higher priority HI-criticality jobs. That is, a job generated by task τ_i in set S_j will yield to all jobs generated by tasks in sets $S_0, S_1, .., S_{j-1}$. There are three tasks at each priority level and the scheduler can arbitrarily pick between jobs of these tasks. In addition, each job of HI-criticality task τ_i keeps track of how much LO-criticality work has executed since its release. While this work is smaller than ℓ_i, it yields to LO-criticality jobs. After this work is equal to ℓ_i, it never yields to LO-criticality jobs.

HI-criticality mode scheduling algorithm.   The system transitions to HI-criticality mode at time T_r if any HI-criticality job executes for C_i^L time without signalling completion. After T_r, the jobs generated by LO-criticality task are never given any execution time and the HI-criticality jobs just run with earliest deadline first scheduling.

4.3   Proof of Schedulability

We must now show that this task system is always schedulable using the scheduling algorithm described above. The basic idea is that the scheduler tries to mimic the periodic schedule.

Recall that, in the periodic schedule in LO-criticality mode, jobs generated by tasks in S_0 never experience any interference since they are executed as soon as they are released. In general, jobs of tasks of sets $S_0, S_1, ..., S_{j-1}$ execute before jobs of tasks in S_j; in addition jB jobs of the LO-criticality task can execute before tasks in S_j. Therefore, tasks in S_j are guaranteed to complete by time $(2j+1)B$ in the periodic schedule. The scheduler described above is specifically designed to ensure that a job of τ_i in set S_j never experiences more interference in the sporadic schedule than it experiences in the periodic schedule; therefore, the response time of each HI-criticality task in LO-criticality behavior is at most its response time in the strictly periodic scheduler. This leads to two good properties. First, and more obviously, all HI-criticality tasks meet their deadlines in LO-criticality mode. Second, and less obviously, since the response time of all HI-criticality jobs in LO-criticality mode is bounded by their response time in the periodic schedule, the transition point occurs "early enough." That is, if a HI-criticality job exceeds its LO-criticality WCET and the system transitions into HI-criticality behavior, then all pending jobs still have enough time to complete their

HI-criticality work C_i^L. In addition, we must argue that while the system remains in LO-criticality mode, all LO-criticality tasks also meet their deadlines. This is possibly the most counter-intuitive part of the argument; here we argue that the slack condition ensure that at most a total of B HI-criticality work reaches slack 0 and preempts any single LO-criticality job during its execution.

We will first argue that all jobs (from both HI-criticality and LO-criticality tasks) meet their deadlines while the system remains in LO-criticality behavior and then we will argue that HI-criticality jobs meet their deadlines if the system transitions to HI-criticality behavior.

4.3.1 Correctness Proof for LO-Criticality Behavior

We will first argue that all HI-criticality tasks meet their deadlines in LO-criticality behavior. This is relatively straightforward. We first prove a simple lemma about *idle instant* – that is, an instant such that all pending work has completed. In particular, at the idle instant, all the jobs that were released before this instant has completed.

▶ **Lemma 3.** *While the system remains in* LO-*criticality behavior, in any interval of size* $2mB$*, there is at least one idle instant.*

Proof. We start at an idle instant t and argue that the next idle instant is within $2mB$ time. Look at the interval from t to $t + 2mB$. During this interval, at most mB LO-criticality work is released and at most mB HI-criticality work is released. If this interval stays busy, then all this work is done by the end of this interval; therefore, there is an idle instant at time $t + 2mB$ if there is not one before. ◀

The following corollary follows from Lemma 3 and the fact that the inter-arrival time between consecutive jobs of the same HI-criticality task in our task system is at least $2mB$.

▶ **Corollary 4.** *While the system remains in* LO-*criticality mode, between any two consecutive job releases of the same* HI-*criticality task, there is an idle instant.*

We can now use this corollary to show that all HI-criticality tasks meet their deadlines in LO-criticality mode.

▶ **Lemma 5.** *The response time of a job of task τ_i in set S_j is bounded by $(2j + 1)B$ in* LO-*criticality mode. Therefore, all* HI-*criticality tasks meet their deadlines in* LO-*criticality mode.*

Proof. Due to Corollary 4, no job can suffer interference from two jobs of the same task since there is an idle instant between consecutive releases. A HI-criticality task τ_i in set S_j has priority j and only tasks in sets $S_0, S_1, ..., S_j$ can interfere with it. Therefore, the total LO-criticality WCET of higher priority tasks including its own priority, is exactly $(j + 1)B$. In addition, the total interference it can experience from LO-criticality tasks is at most jB (by enforcement of the slack scheduling policy). Therefore, the job has a response time of at most $(2j + 1)B$. ◀

We now prove the more interesting result that all jobs generated by the LO-criticality task also meet their deadlines in the LO-criticality mode. This argument depends on the slacks. In general, one might worry that if there are many HI-criticality jobs with small remaining slack, then a LO-criticality job may starve and not be able to get enough computation time by its deadline. We will now argue that this cannot happen.

Intuitively, this is easiest to see in a strictly periodic schedule starting at time 0. The jobs of tasks in S_0 have no slack since for all these tasks $\ell_i = 0$ – therefore, they execute

first taking up the first B time steps. After this, however, the first LO-criticality job gets to execute and meets its deadline. At this point, jobs in S_1 have exhausted their slack and they execute next. The jobs in S_2 started with slack of $2B$ and have B slack remaining, so the second LO-criticality job executes before them. Therefore, the LO-criticality job gets B execution in the interval from its release time to its deadline every time and meets all its deadlines. We will do an induction argument to see this. In order to do this induction, we need two additional definitions.

We define $w(\tau_i, t)$ as the remaining work of task τ_i at time t. At an idle instant, $w(\tau_i, t) = C_i^L$. If no job of a task τ_i has been released since the last idle instant or a job has been released, but has not yet done any execution, then $w(\tau_i, t) = C_i^L$. As a job of a task executes, its w decreases. In particular, if a job of the task τ_i has done work w since the last idle instant, then $w(\tau_i, t) = C_i^L - w$. If the job (released since last idle period) has completed, then $w(\tau, t) = 0$.

We will define a quantity *running slack* $r\ell_i$ for each task τ_i. At the beginning of the execution and at any idle instant, the running slack $r\ell_i$ is set to the task slack ℓ_i – that is, tasks in set S_j get a slack of jB. This running slack remains unchanged until a job of this task τ_i is released. Once a job J_i of a task τ_i is released, $r\ell_i$ decreases every time step that a job of the LO-criticality task executes – this corresponds to keeping track of how much LO-criticality work has executed since job J_i arrived. By the scheduler definition, once $r\ell_i = 0$ the job J_i stops yielding to LO-criticality jobs and no LO-criticality job can execute until J_i finishes executing. Note that it is sufficient to reset $r\ell_i$ values at each idle instant since by Corollary 4, there is an idle instant between consecutive arrivals of any HI-criticality task; therefore, the running slack $r\ell_i$ is always reset to ℓ_i between consecutive job arrivals of the same task. (Note that since we have a work-conserving scheduler, HI-criticality jobs with slack larger than 0 can run if there is no LO-criticality pending job.)

Recall that we must argue that if a LO-criticality job arrives at time t, at most B HI-criticality work can preempt it – this would ensure that the LO-criticality job gets enough execution time within its release time to deadline scheduling window of $[t, t+2B]$ to complete its own execution requirement of B. We define a quantity that generalizes this notion: we define $Ł(K, t)$ as the total work with slack at most K after time step t. In other words, $Ł(K, t) = \sum_{r\ell_i \leq K} w(\tau_i, t)$. If $Ł(K, t) = W$, then W HI-criticality work has slack smaller than K at time t. At any time instant t, some HI-criticality job can preempt a LO-criticality job only if its work is in $Ł(0, t)$.

At a high level, the function $Ł(K, t)$ can be looked upon as a measure of how much work is urgent (and to what extent). Consider the platform at time r^J when a LO-criticality job J arrives. All the work in $Ł(0, r^J)$ has no slack (its $r\ell_i$ value is 0) and will execute before this LO-criticality job J can start execution. After this the LO-criticality job J can start, but all tasks that have pending jobs in the system will reduce their $r\ell_i$ values – therefore, when J has completed 1 unit of work at time $r^J + t_1$, the pending work that was in $Ł(1, r^J)$ (had running slack $r\ell_i$ smaller than 1 at time t) will now be in $Ł(0, r^J + t_1)$ and will preempt this LO-criticality job. In general, if J has completed b work by time $r^J + t_b$, then the work that was in $Ł(b, r^J)$ may now be in $Ł(0, J_r + t_b)$ and can preempt this LO-criticality job at time $r^J + t_b$. However, note that any work that was not in $Ł(B - 1, r^J)$ cannot become urgent before J completes since its slack can only reduce by B and therefore, cannot become 0 before J completes.

We can divide this $Ł$ function into two components: $Ł_f$ and $Ł_p$. $Ł_f$ contains the work of all the tasks where no job of this task has yet arrived into the system since the last idle

instant. Therefore,

$$Ł_f(K,t) = \sum_{r\ell_i \leq K, \tau_i \text{ not pending}} w(\tau_i,t) \quad = \sum_{r\ell_i \leq K, \tau_i \text{ not pending}} C_i^L.$$

$Ł_p$ contains the work of all the jobs that are pending. For all tasks whose jobs have finished executing since the previous idle instant, their corresponding w is 0. Therefore, for all parameters, $Ł_f$ and $Ł_p$ sum up to $Ł$.

We first prove an invariant on $Ł_f$.

▶ **Lemma 6.** *At any time t, $Ł_f(jB-1,t) \leq jB$ for all $1 \leq j \leq m$.*

Proof. At an idle instant, at time 0, before any jobs arrive, all running slacks reset. Therefore, by definitions of ℓ_i, the tasks in set S_j have $r\ell_i = jB$. Therefore, there is B total work with slack 0, B work with slack B, and so on. Therefore, the total work with slack $B-1$ is $Ł_f(B-1,0) = B$, the total work with slack $2B-1$ is $Ł_f(2B-1,0) = 2B$ and, in general, the total work with slack at most $jB-1$ is $Ł_f(jB-1,0) = jB$. Since the total pending HI-criticality work at any time is at most mB and nothing has slack more than $(m-1)B$, we have $Ł_f((m-1)B,t) = Ł_f(mB-1) \leq mB$. $Ł_f$ is maximum at an idle instant, since $Ł_p$ is 0 for all values. After this point, $Ł_f$ only reduces as jobs arrive in the system.   ◀

We will now prove an invariant on the function $Ł$ which will let us prove that the LO-criticality task is schedulable.

▶ **Lemma 7.** *When a LO-criticality job J arrives at time r^J, we have $Ł(jB-1,r^J) \leq jB$ for all $1 \leq j \leq m$.*

Proof. We will prove a stronger statement by induction.[7] Without loss of generality, we reset time to 0 at an idle instant. Say an LO-criticality job arrives at time r^J. After the arrival of this LO-criticality job and until its completion or until an idle instant (whichever is sooner), we have to prove the following: Say at time $r^J + b + c$, the LO-criticality job has executed for b time steps and some HI-criticality jobs have executed for c time steps. Then, we have $Ł(jB-1-b,r^J+b+c) \leq jB-c$. If we prove this statement, then we obviously get the lemma by substituting $b = c = 0$.

At an idle instant, at time 0, all running slacks reset and all L is in $Ł_f$. Therefore, Lemma 6 gives us the base case.

We now induct on time steps. Say a LO-criticality job J was released at time step r^J and has a deadline of d^J. Lets say that at in interval $[r^J, r^J + b + c]$, the schedule has done c HI-criticality work and b work on this LO-criticality job. By the inductive hypothesis, we have $Ł(jB-1-b,r^J+b+c) \leq jB-c$.

Case 1: On the next step, say we do 1 unit of LO-criticality work. Then at time step $r^J+b+c+1$, we have done $b+1$ LO-criticality work and c HI-criticality work. We must show that $Ł(jB-1-b-1,r^J+b+c+1) \leq jB-c$. Now the slack of all jobs that have arrived by time r^J+b+c, but not completed reduces by 1. Therefore, we have $Ł_p(iB-1-b-1,r^J+b+c+1) = Ł_p(iB-1-b,r^J+b+c)$. In addition, $Ł_f(iB-1-b-1,r^J+b+c) = Ł_f(iB-1-b,r^J+b+c)$ since $b \leq B$ and $Ł_f$ only changes at discrete intervals of size B. Therefore, we have $Ł(iB-1-b-1,r^J+b+c+1) = Ł_p(iB-1-b-1,r^J+b+c+1)+Ł_f(iB-1-b-1,r^J+b+c+1) =$

---

[7] Without loss of generality, we are assuming discrete time for ease in induction. One can imagine that each time step takes as long as a machine instruction. One can also do this induction based on events; but it gets more complicated.

$Ł_p(iB - 1 - b, r^J + b + c) + Ł_f(iB - 1 - b, r^J + b + c) \le jB - c$. If a job is released at time step $r^J + b + c + 1$, this moves its work from $Ł_f$ to $Ł_p$, but does not change the overall sum in $Ł$.

Case 2: On the next step, we do one unit of work on some HI-criticality job J_i from task τ_i in set S_j. The only way we can do this while job J is still pending is because $r\ell_i$ has become 0. Therefore, $w(\tau_i)$ is part of $Ł(0, r^J + b + c)$. Since $Ł$ function is cumulative, this work is also part of all $Ł(K, r^J + b + c)$ for all $K > 0$. Therefore, when this work is executed, all $Ł$'s reduce by 1 on this time step. Therefore, $Ł(jB - 1 - b, r^J + b + c + 1) = Ł(jb - 1 - b, r^J + b + c) - 1 \le jB - c - 1$.

The deadline of the LO-criticality job is $2B$ time steps after its release time. Say we did $W \le B$ LO-criticality work by the deadline and $2B - W$ HI-criticality work. Therefore, we have $Ł(iB - 1 - W, J_d) \le iB - 2B + W$. Since $W \le B$, we have $Ł(iB - 1 - B, J_d) \le iB - 2B + W \le iB - B$. Since this invariant is true for all $0 \le i \le m - 1$, we have, $Ł(iB - 1, J_d) \le iB$. If the next job is immediately released, we have satisfied the invariant. If the next job is not immediately released, then some HI-criticality job executes and the $Ł(iB - 1, t)$ for current time instant t only decreases, therefore, the invariant remains true when the next LO-criticality job is released. ◄

We can now argue that all LO-criticality jobs meet their deadlines if the system remains in LO-criticality mode. Recall that we argued that any work that was not in $Ł(B - 1, t)$ when a LO-criticality job J arrives at time t cannot become urgent before J completes since its slack cannot become 0 before J completes.

▶ **Lemma 8.** LO-*criticality jobs finish by their deadlines in* LO-*criticality mode.*

Proof. From Lemma 7, we know that when a LO-criticality job arrives at time t, $Ł(B - 1, t) \le B$. Therefore, at most B work has slack smaller than B when this job arrives. As this job executes, slack of all jobs reduces, but only the work that already has slack smaller than B can get to 0 before this job finishes. Therefore, only this B work can interfere with this LO-criticality job. Therefore, the LO-criticality job gets to execute for B time units within its scheduling window of $2B$ and hence does not miss its deadline. ◄

4.3.2 Correctness for HI-Criticality Behavior

We now have to show that all jobs can meet their deadlines in HI-criticality mode. In particular, we will say that a transition occurs at T_r when some HI-criticality job executes for C_i^L time steps, but does not signal completion. The LO-criticality task does not get any further execution after time T_r. For all jobs that are pending at time T_r or arrive after T_r, we must argue that they can complete by their deadlines even if they execute for HI-criticality WCET C_i^H.

The intuition behind this proof relies on the periodic schedule. Recall that in the periodic schedule's schedulability relies on the following fact: if a job J from task τ_i in set S_j is released at time r^J (with deadline $d^J = r^J + 2m$), then it is guaranteed to finish its LO-criticality work by time $r^J + (2j + 1)B$. Therefore, if this job were going to cause a mode-transition, then it would happen before this time, implying $T_r = r^J + (2j + 1)B$ (or earlier). In addition, by this time there are no pending jobs from any set in $S_1, S_2, ..., S_{j-1}$. Therefore, we can only have pending jobs from tasks in sets $S_j, S_{j+1}, ..., S_{m-1}$. The total HI-criticality WCET for all jobs in this set is $2(m - j)B$ and we have already done B work from set S_j. Therefore, the total pending work is $(2m - 2j - 1)B$. Since all HI-criticality jobs have the same release time and deadline in the periodic schedule, all this work can be completed by the deadline. Staring

from the next hyper-period, we have an implicit deadline task system with on HI-criticality tasks which have total utilization of 1; therefore, they are schedulable.

In the sporadic schedule, we no longer have the nice property that all pending HI-criticality jobs have the same deadline. However, we can still argue the crucial point that the total pending work at the transition time T_r is bounded. In particular, just like the periodic schedule, if a job from task τ_i from set S_j causes the transition, then no job from sets $S_0, S_1, ..., S_{j-1}$ can be pending. Therefore, the total amount of pending work is bounded.

We will say that there is carry-over work for task τ_i if a job J of task τ_i has been released by the transition time T_r, but has not completed. We denote the set of carryover jobs by C. We say that each task τ_i has completed d_i work by transition time T_r if the latest release of this job has done d_i work by time T_r. This quantity d_i is defined for both carryover jobs and non-carryover jobs.

We now prove a generalization of Lemma 5.

▶ **Lemma 9.** *Consider a job J_i of task τ_i in set S_j with release time $r^{J_i} < T_r$ which has not completed by time T_r. We have $T_r \leq r^{J_i} + 2jB + \sum_{\tau_k \in S_j} d_k$. Therefore, the deadline for job J_i is $d^{J_i} \geq T_r + 2mB - 2jB - \sum_{\tau_k \in S_j} d_k$.*

Proof. After r^{J_i}, only tasks of higher priority than τ_i (those belonging to S_p where $p < j$) tasks in S_j and LO-criticality tasks can execute before. Since the system remains in LO-criticality mode until time T_r, no job has previously exceeded its LO-criticality WCET. There is only jB work with higher priority and due to the slack condition, only jB LO-criticality work can execute. Finally, $\sum_{\tau_k \in S_j} d_k$ work was done for tasks in set S_j. Since the system stays busy between r^{J_i} and T_r, we get the condition on release time. The condition on d^{J_i} is obtained by adding $2mB$ (the relative deadlines of all HI-criticality tasks) to r^{J_i}. ◀

Lemma 9 provides us a crucial condition that no pending job's deadline is too close to the transition point. Even more crucially, the higher the priority of the pending job (the smaller the j value), the farther in the future its deadline is guaranteed to be. This is due to the following. Either this job just completed its LO-criticality WCET C_i^L at time T_r and did not signal completion (causing the mode transition) or this job has not yet completed its LO-criticality WCET C_i^L. (If it had completed its C_i^L earlier, it cannot still be pending since it would either complete or cause a transition earlier.) But we know that higher priority jobs have a smaller response time in LO-criticality mode. Therefore, if higher-priority jobs were going to exceed their LO-criticality WCET, causing a transition, then they cause this transition sooner after their release compared to lower priority jobs.

This condition helps us prove completion of carryover jobs using a *reverse-priority* scheduler. This scheduler only looks at carry-over jobs (it ignores all jobs that arrive after T_r) and schedules jobs in reverse order of priorities. That is, carryover jobs from tasks in set S_p where $p > j$ are scheduled before carryover jobs from set S_j for all j.

▶ **Lemma 10.** *All jobs with carry-over work complete by their deadlines using the reverse-priority scheduler.*

Proof. As we mentioned earlier, Lemma 9 implies that if we look at all the pending jobs at transition time T_r, the higher the priority of the job, the farther in the future the deadline of the job is guaranteed to be. This does not mean that all deadlines are ordered by priorities – it only means that the higher priority jobs cannot have deadlines that are too soon after T_r.

It is easy to see that with the reverse priority scheduler, that carry-over jobs with priority j will finish by time $T_r + \sum_{\tau_k \in S_j, S_{j+1}, ..., S_{m-1}} (C_k^H - d_k) \leq T_r + 2(m-j)B - \sum_{\tau_k \in S_j, S_{j+1}, ..., S_{m-1}} d_k$ time using the reverse priority scheduler. Since the deadline of any job J_i with priority j is $d^{J_i} \geq T_r + 2(m-j)B - \sum_{\tau_k \in S_j} d_k$ (Lemma 9), the job meets its deadline. ◀

▶ **Corollary 11.** *All jobs with carry-over work complete by their deadlines using EDF.*

Proof. Since all carry-over jobs were released before T_r, EDF will always schedule them before scheduling any jobs released after T_r. Therefore, just like the reverse-priority scheduler, EDF ignores all non-carryover jobs until the carryover work is done. We know that EDF is an optimal scheduler and if any scheduler can schedule a set of jobs, EDF can. Therefore, since reverse-carryover scheduler and EDF consider the same set of carryover jobs, and reverse-carry over scheduler meets all deadlines (Lemma 10), then EDF also meets all deadlines. ◀

We must now prove that all jobs that are released after transition time T_r can meet their deadlines. These jobs are easier to reason about since after T_r, we do not have to worry about mixed-criticality execution any more – we simply have a simple implicit deadline task system where all tasks have the same period and deadlines.

▶ **Lemma 12.** *All* HI-*criticality jobs meet their deadlines in* HI-*criticality mode when scheduled with EDF.*

Proof. Corollary 11 indicates that all carryover jobs meet their deadlines. So we need only worry about jobs that are released after time T_r. Since all jobs have the same relative deadline and we schedule using EDF, no job can interfere with a job that is released after itself. Therefore, no job can suffer interference from 2 jobs of the same task. Since the total HI-criticality WCET of all HI-criticality tasks collectively is $2mB$ and the relative deadline is $2mB$, there is enough time to execute one job of all tasks after a job's release time and still meet its deadline. ◀

Lemmas 5, 8 and 12, and Theorem 2 collectively allow us to conclude that given an instance I of 3-PARTITION, we can generate a sporadic task system that is schedulable if and only if I has a solution. Therefore, a dual-criticality schedulability for sporadic task systems is NP-hard in the strong sense – this completes the proof of the main result in this section.

5 Conclusions and Discussion

Designing efficient mixed-criticality scheduling algorithms is a very challenging problem. In this paper we have described our efforts at approach this problem by breaking it out into two constituent components, seeking to separate the difficulties that arise from the on-line nature of mixed-criticality scheduling – the fact that much important information is simply not known prior to run-time – and those arising from computational complexity issues. As a major step to doing so, we first needed to establish that determining MC-schedulability for recurrent (periodic or sporadic) implicit-deadline task systems is NP-hard in the strong sense; in this manner, we showed that all three mixed-criticality scheduling problems considered – scheduling collections of jobs, of periodic tasks, and sporadic tasks – have to deal with both sources of intractability.

With regards to the scheduling of collections of independent jobs, we established, for the first time, an approximation ratio for OCBP vis-à-vis MC-schedulability, thereby quantifying OCBP's deviation from optimal behavior due to computational complexity issues (as opposed to its sub-optimality due to non-clairvoyance). This result gave rise to several interesting issues and questions:

1. A somewhat odd aspect of this result is that OCBP's approximation ratio and its competitive ratio (ie., its performance hit vis-à-vis a clairvoyant scheduler) are both equal to the same constant ($\Phi, \approx 1.618$); we do not have an understanding as to why this should be so.

2. We observed that speedup factor, when used as a metric for both competitive ratio and approximation ratio, does not appear to compose in any meaningful manner: while the competitive ratio of any MC-scheduling job algorithm is $\geq \Phi$ and OCBP's approximation ratio for solving the MC-scheduling problem is Φ, the speedup factor of OCBP vis-à-vis clairvoyant schedulability is also Φ (rather than, say, 2Φ or Φ^2).

3. It would be interesting to seek to characterize other algorithms that have been proposed for scheduling dual-criticality job instances (such as MC-EDF [13] and LE-EDF [9, page 29]) by approximation ratios. These algorithms have been experimentally observed to perform better than OCBP on randomly-generated data; perhaps their superiority can be quantified by showing that they have a smaller approximation ratio than Φ?

With regards to EDF-VD and the scheduling of recurrent task systems, the non-optimality of EDF-VD vis-à-vis MC-schedulability had not, to our knowledge, been explored previously. By showing that MC-schedulability for dual-criticality recurrent task systems is NP-hard in the strong sense, we have provided some justification for the use of these non-optimal algorithms. There are several open issues concerning the analysis of EDF-VD – we would, in essence, like to eventually have as complete an understanding of EDF-VD's effectiveness as we have currently been able to obtain for OCBP. This includes determining whether EDF-VD is optimal (vis-à-vis MC-schedulability – it is known to not be optimal vis-à-vis clairvoyant schedulability) for certain classes of task systems, determining approximation ratios, etc.

References

1 N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, The University of York, England, 1991.

2 N. C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems.* PhD thesis, Department of Computer Science, University of York, 1993.

3 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 *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

4 Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE, April 2010.

5 Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

6 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, 2017. `doi:10.1145/3131347`.

7 Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I. Davis. On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:25, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2017.9`.

8 R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

9 Zhishan Guo. *Real-Time Scheduling Of Mixed-Critical Workloads Upon Platforms With Uncertainties.* PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 2016.

10   Haohan Li. *Scheduling Mixed-Criticality Real-Time Systems*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 2013.

11   C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

12   D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

13   Dario Socci, Petro Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, Paris (France), 2013. IEEE Computer Society Press.

14   J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

15   Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.