The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems*

Zhishan Guo
The University of North Carolina at Chapel Hill
201 S Columbia Street
Chapel Hill, North Carolina 27599
zsguo@cs.unc.edu

Sanjoy Baruah
The University of North Carolina at Chapel Hill
201 S Columbia Street
Chapel Hill, North Carolina 27599
baruah@cs.unc.edu

ABSTRACT

Most prior work on mixed-criticality (MC) scheduling has focused on a model in which multiple WCET parameters are specified for each job, the interpretation being that the larger values represent "safer" estimates of the job's true WCET. More recently, a different MC model has been studied in which it is assumed that the precise speed of the processor upon which the system is implemented varies in an *a priori* unknown manner during runtime, and estimates must be made about how low the actual speed may fall.

The research reported in this paper seeks to integrate the varying-speed MC model and the multi-WCET one into a unified framework. A general model is proposed in which each job may have multiple WCETs specified, *and* the precise speed of the processor upon which the system is implemented may vary during run-time. We reinterpreted the key idea behind the table-driven MC scheduling scheme proposed in one of our recent work, and provide a more efficient algorithm named LE-EDF. This algorithm strictly generalizes algorithms that were previously separately proposed for MC scheduling of systems with multiple WCETs as well as for MC scheduling on variable-speed processors. It is shown that LE-EDF outperforms (via simulation) and/or dominates existing algorithms (under theoretical proof). LE-EDF is also compared with optimal clairvoyant algorithm using the metric of speedup factor.

1. INTRODUCTION

Special-purpose processors used in implementing safety-critical systems are designed to be highly predictable: given the specifications of the workload that is to be executed upon such a processor, it is possible to provide tight bounds on the worst-case run-time behavior of the system during system design time, to a very high level of assurance. Such design-time predictability is essential for safety-critical functionalities, but is difficult to achieve with Commercial Off-The-Shelf (COTS) processors that are typically engineered to provide good average-case performance rather than worst-case guarantees.

Estimating WCET. The worst-case execution time (WCET) of a given piece of code upon a specified platform represents an upper bound of the duration of time needed for it to execute. The WCET abstraction plays a central role in the analysis of real-time systems.

Determining the exact WCET of an arbitrary piece of code is provably an undecidable problem. Even when severe restrictions are placed upon the structure of the code (e.g., loop bounds must be known at compile time), sophisticated features that are found upon COTS processors that are used in embedded systems today (such as multi-level cache, deep pipelining, speculative out-of-order execution, etc.) are hard to analyze and make it extremely difficult to determine WCET precisely. Devising analytical techniques for obtaining tight upper bounds on WCET is currently a very active and thriving area of research, and sophisticated tools incorporating the latest results of such research have been developed (see [17] for an excellent survey).

A large body of prior research on *mixed criticality* (MC) scheduling (see [6] for a review of some of this work) has focused upon dealing with the phenomenon that multiple WCET bounds may be provided by different WCET analysis tools. Although it may be necessary (for instance, mandated by a statutory Certification Authority) to use the most conservative WCETs for validating the correctness of safety-critical functionalities, less conservative WCETs should suffice for validating the correctness of less critical functionalities. Such uncertainty is modeled by assigning each job two WCET parameters – a larger, more conservative one, and a smaller, less conservative one. Some of the jobs are designated as being safety-critical; the remaining ones are not safety-critical. The objective is to determine a run-time scheduling strategy to ensure that (i) all jobs complete by their deadlines if each job completes execution upon having executed for no more than the smaller of its WCET values; and (ii) the jobs designated as being safety-critical continue to complete by their deadlines (although the non-critical jobs may fail to do so) if some job does not complete execution upon having executed for up to the smaller of its WCET values, although each job does complete upon having executed for the larger of its WCET values.

Varying-speed processors. Recently, some efforts have been made (see, e.g., [9, 3]) to address another source of modeling uncertainty inherent in characterizing the run-time behavior of real-time systems (although not uncertainty in WCET estimation alongside). Such modeling re-

*Work supported by NSF grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors Corp.

RTNS 2015, November 04-06, 2015, Lille, France
 ISBN 978-1-4503-3591-1/15/11 $15.00
DOI: http://dx.doi.org/10.1145/2834848.2834852

quires that some assumptions be made about the run-time behavior of the processor upon which the code is to execute; for example, the *clock speed* of the processor during run-time must be known in order to be able to determine the rate at which instructions will execute. However, conditions during run-time, such as changes to the ambient temperature, the supply voltage, etc., may result in variations in the clock speed — for instance, a system programmer may use the userspace Linux command `cpuspeed` to configure a system to reduce CPU clock speeds if the core temperature gets too high. Processor speed could additionally change during run-time in embedded devices that use dynamic frequency scaling in order to reduce energy consumption. At the hardware level, too, innovations in computer architecture for increasing clock frequency can lead to varying-speed clocks during run-time: e.g., [5] describes a recently-introduced technique for detecting whether signals are late at the circuit level within a CPU micro-architecture, and if so to recover by delaying the next clock tick so that logical faults do not propagate to higher (i.e., the software) levels.

In order to be able to guarantee that deadlines are met under all run-time conditions, conservative analysis prior to run-time must make the most pessimistic assumptions regarding clock speed: that during run-time *the clock speed takes on the lowest possible value*. If this lowest possible value is highly unlikely to be reached in practice during actual runs, then a significant under-utilization of the CPU's computing capacity will be observed during run-time. A less conservative analysis, on the other hand, may assume a less pessimistic (i.e., larger) lower bound on the clock speed during run-time. Using such an assumption as the basis for making resource-allocation decisions will likely lead to more efficient usage of the CPU's computing capacity during run-time; however, there is a possibility that the actual processor speed will fall below the lower-bound estimate used in the analysis (thereby invalidating the conclusions drawn during such analysis).

Contribution and Organization. This paper provides a generalized MC framework (in Sec. 2) that covers the uncertainties arising from *both* upper bounds of WCET estimations *and* lower bounds on processor speeds, while prior work only considers one dimension of such uncertainty. In Sec. 3, an existing table-driven algorithm (for varying-speed MC scheduling) is reinterpreted with simpler design that follows EDF execution during run-time, named LE-EDF. In Sec. 4, we show its *optimality* theoretically under the single WCET case, while for the general case, we compare LE-EDF with optimal clairvoyant algorithm with the metric of speedup bound, as a function of the set's parameters (*loads*), with upper bound of 4/3. In Sec. 5, we further show that under the constant speed case, LE-EDF *strictly dominates* the OCBP algorithm [4] (which is a key result for MC job scheduling). Experimental study is also provided to support such result, and further suggests the domination over the MCEDF algorithm [16] (which serves as the current state of the art). We conclude in Sec. 6 by discussing extensions that we are currently working upon, and by placing this work within a larger context of the mixed-criticality scheduling theory.

2. MODEL AND DEFINITIONS

In this paper, we model a mixed-criticality real-time workload as being comprised of basic units of work known as mixed-criticality *jobs*. Finite collections of independent jobs may arise in frame-based approaches of real-time scheduling, while some real-time system have only periodic tasks with a small hyperperiod that could also be represented as a collection of jobs.

Each MC job J_i is characterized by a 4-tuple of parameters: a release time a_i, a vector $\langle c_i(\text{LO}), c_i(\text{HI})\rangle$ of two WCET values where $c_i(\text{LO}) \leq c_i(\text{HI})$ for HI-criticality jobs and $c_i(\text{LO}) = c_i(\text{HI})$ for LO-criticality ones, a deadline d_i, and a criticality level[1] $\chi_i \in \{\text{LO}, \text{HI}\}$.

A mixed-criticality *instance* $\mathcal{I}$ is specified by

- a collection of MC jobs: $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, and
- a processor that is characterized by two thresholds: a *normal* speed s_ν and a *degraded* speed $s_\delta (\leq s_\nu)$.

The interpretation is that the jobs in $\mathcal{J}$ are to execute on a single shared preemptive processor that has two modes: a *normal* mode and a *degraded* (or *faulty*) mode. In normal mode, the processor executes as a speed-s_ν (or faster) processor and hence completes at least s_ν units of execution per time unit, whereas in degraded mode it completes less than s_ν, but at least s_δ units of execution per time unit. The processor starts out executing at or above its normal speed, and it is not *a priori* known how the processor speed will vary during run-time.

DEFINITION 1. *A scheduling strategy for MC instances is* ***correct*** *if upon scheduling any MC instance* $\mathcal{I} = (\{J_1, J_2, \ldots, J_n\}, s_\delta, s_\nu)$*, it satisfies the following two properties P1 and P2.*

P1. Each job J_i meets its deadline if all jobs complete execution upon having executed for no more than their LO*-criticality WCETs,* and *the processor speed remains $\geq s_\nu$ throughout Interval $[a_i, d_i)$; and*

P2. Each HI*-criticality job J_i meets its deadline if all* HI*-criticality jobs complete execution upon having executed for no more than their* HI*-criticality WCETs, and the processor speed remains $\geq s_\delta$ throughout Interval $[a_i, d_i)$;*

A scheduling strategy for MC instances is ***partially correct*** *if it satisfies P2 above, but not necessarily P1.*

That is, a partially correct scheduling strategy ensures the correct execution of HI-criticality jobs provided the processor executes at or above its degraded speed and each HI-criticality job completes upon executing for no more than its HI-criticality WCET. A correct scheduling strategy additionally ensures the correct execution of LO-criticality jobs if the processor executes at or above its normal speed and each job completes upon executing for no more than its LO-criticality WCET.

A *clairvoyant scheduling algorithm* is one that knows, prior to scheduling an instance, (i) precisely how much execution time each job in the instance will require in order to complete, and (ii) the precise manner in which the processor speed will vary during run-time.

[1] In common with much prior work in MC-scheduling, we start out considering systems with just two criticality levels specified. Generalization to systems with more than two criticality levels is discussed in Section 6.

DEFINITION 2 (***optimal scheduling strategy***). *An optimal scheduling strategy for MC instances possesses the property that if it fails to maintain correctness (partial correctness, respectively) for a given MC instance $\mathcal{I}$, then no non-clairvoyant algorithm can ensure correctness (partial correctness, resp.) for the instance $\mathcal{I}$.*

We now state some conventions adopted in this paper, as well as some further notation. Without loss of generality, we will assume that the HI-criticality jobs in given MC instance $\mathcal{I}$ are indexed $1, 2, ..., n_h$ and the LO-criticality jobs are indexed $n_{h+1}, ..., n$. Let $t_1, t_2, ..., t_{k+1}$ denote the at most $2n$ distinct values for the release time and deadline parameters of the n jobs, in strictly increasing order (redundancy is eliminated, so $\forall j, t_j < t_{j+1}$). These release time and deadlines partition the time duration $[\min_i\{a_i\}, \max_i\{d_i\})$ into k *intervals*, which will be denoted as $I_1, I_2, ..., I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

3. ALGORITHM LE-EDF

In this section we describe Algorithm LE-EDF[2] for scheduling MC instances that are represented using the model discussed in Section 2 above. We will also illustrate, via a running example, the behavior of LE-EDF when scheduling such an MC instance.

The high-level description of our algorithm is as follows. Given an MC instance $\mathcal{I} = (\mathcal{J}, s_\nu, s_\delta)$, similar to the algorithm described in Sec. 4 of [10], we first construct, prior to run-time, a *scheduling table* that reserves a certain amount of execution time for each HI-criticality job within each time interval $I_j = [t_j, t_{j+1})$, for $1 \leq j \leq k$, in order to ensure that no HI-criticality deadline will be missed even under the most conservative case. To this end, LE-EDF is in some sense similar to the zero-slack technique developed by Niz et al. [15], which mainly focused on fixed priority schemes such as rate-monotonic instead of EDF based ones (which is our focus). To comply with this scheduling table, HI-criticality jobs are divided into *sub-jobs* with different deadlines. Dispatch decisions at run-time are taken in a manner that HI-criticality jobs being executed for at least the amounts mandated in the scheduling table (by having sub-jobs meeting their assigned deadlines), while using the remaining computing capacity to execute LO-criticality jobs. The latest execution (LE) manner in which the sub-job set is constructed is described in Section 3.1; run-time dispatching (under EDF) is detailed in Section 3.2.

3.1 Sub-job Construction (LE)

To construct the scheduling table, we first identify (Step 1 below) the latest time intervals during which the HI-criticality jobs must execute if (i) they are to complete execution on a degraded processor, and (ii) each were to execute for its HI-criticality WCET; having identified these intervals, we construct (in Step 2) an EDF schedule for the HI-criticality jobs in these intervals.

Step 1. *Considering only the HI-criticality jobs in the instance, determine the intervals during which the jobs would execute upon a speed-s_δ processor, if*

1. *each job executes for its HI-criticality WCET,*
2. *each job completes by its deadline, and*
3. *execution occurs as late as possible.*

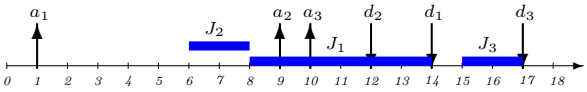
It is evident that these intervals may be determined by filling in the schedule "backwards"; i.e., considering the jobs in non-increasing order of their deadlines, and allocating the cumulative execution requirements of these jobs. They can therefore be determined in $\mathcal{O}(n_{\text{HI}} \log n_{\text{HI}})$ time (which is the complexity of sorting), where n_{HI} denotes the number of HI-criticality jobs. We illustrate this in Example 1 below.

EXAMPLE 1. *Throughout this section we will consider the instance consisting of the six jobs J_1–J_6 shown in tabular form in Figure 1, that is to be implemented upon a preemptive processor of normal speed $s_\nu = 1$ and degraded speed $s_\delta = 0.5$.*

J_i	a_i	c_i(LO)	c_i(HI)	d_i	χ_i
J_1	1	2	3	14	HI
J_2	9	0.5	1	12	HI
J_3	10	0.5	1	17	HI
J_4	0	7	7	10	LO
J_5	1	0.5	0.5	12	LO
J_6	12	3	3	16	LO

Figure 1: An example MC collection of jobs.

Considering only the HI-criticality jobs J_1–J_3 executing for their HI-criticality WCETs on a speed-s_δ processor, the intervals identified in Step 1 are as follows:



The intervals determined in Step 1 are therefore $[6, 14)$ and $[15, 17)$. (Observe that in this schedule we are only determining execution intervals, not seeking to determine an actual schedule. Hence the fact that job J_2 seems to be "assigned" execution prior to its release time is irrelevant.)

Step 2. *Construct an EDF schedule for the HI-criticality jobs upon a preemptive processor that has speed s_δ during the intervals determined in Step 1 above, and speed zero elsewhere.*

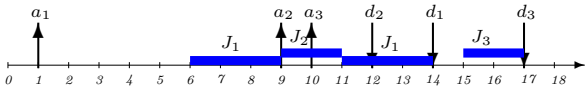
It follows from the optimality property[3] of EDF that if this step fails to ensure that each HI-criticality job receives an execution amount equal to its HI-criticality WCET prior to its deadline, then no scheduling algorithm can guarantee correctness or even partial correctness (see Definition 1), for this instance. We would therefore ***report failure***: this MC instance is not feasible. The remainder of this section, and Section 3.2, assumes that Step 2 above was successful in completing each HI-criticality job prior to its deadline.

EXAMPLE 2. *Consider again the instance of Example 1 that is depicted in Figure 1. In Step 2, the EDF schedule*

[2]The two steps, shown in Sec 3.1, in the construction of the scheduling table explain the name given to our algorithm: *Latest Execution times, with EDF* scheduling

[3]Although the optimality proof of EDF in [8, 12], which is based on a swapping argument, assumes that the processor speed remains constant, it is trivial to extend the proof to apply to processors that are only available during limited intervals, or indeed to arbitrary varying-speed processors.

for the HI*-criticality jobs upon a speed-*0.5 *processor is constructed only within the intervals identified in Step 1; i.e.,* $[6, 14), [15, 17)$[4]*:*



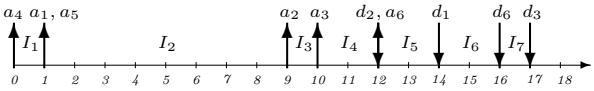
- J_1 *executes during interval* $[6, 9)$ *as the only active job.*
- J_2 *upon release, becomes the earliest-deadline job and is hence allocated execution over the interval* $[9, 11)$*.*
- *Upon* J_2*'s completion,* J_1 *executes during interval* $[11, 14)$ *as the only active job.*
- J_3 *executes in interval* $[15, 17)$ *as the only active job.*

Step 3. *Partition the time-line over* $[\min_i\{a_i\}, \max_i\{d_i\}]$*, and thus the scheduling table, into the* k *intervals* $I_1, I_2, \ldots, I_k$*. (Recall, from Section 2, that these are the intervals defined by the release time and deadlines of all the jobs –* LO*-criticality and* HI*-criticality.) For each* HI*-criticality job* J_i *and each interval* I_ℓ *in which it is scheduled in the EDF schedule constructed in Step 2 above, define a* **sub-job** *of* J_i *with the same release time* a_i*, a WCET equal to the amount of execution that* J_i *is allocated during Interval* I_ℓ*, and a deadline equal to* $t_{\ell+1}$*, the right end-point of Interval* I_ℓ*.*

By dividing HI-criticality jobs into sub-jobs, and setting proper deadlines for them (so that they cannot be suppressed by LO-criticality jobs in the sense of correctness), we are actually mapping the table-driven scheduling derived by [10] into an EDF based schedule, while preserving the optimality (to be shown formally in Sec 4).

Counting the number of sub-jobs. Although an individual job in an EDF schedule for an instance of n jobs may be preempted as many as $(n-1)$ times, it is known (see, e.g., [7]) that the *total* number of preemptions in any EDF schedule for an n-job instance cannot exceed $(n-1)$. From this, it follows that the schedule constructed in Step 2 above will contain no more than $3n_h - 1$ contiguous chunks of execution (here, a $2n_h - 1$ comes from the fact that n_h jobs are being scheduled using EDF, and an additional n_h from the fact that there may be as many as n_h non-contiguous intervals upon which this EDF schedule is executing). Since Step 3 partitions the time-line into no more than $2n-1$ intervals, it follows that the total number of jobs is bounded from above by $3n_h - 1 + 2n - 1$, which is $\mathcal{O}(n)$.

EXAMPLE 3. *For our example instance of Figure 1, Step 3 partitions the time-line into seven intervals* $[0, 1)$*,* $[1, 9)$*,* $[9, 10)$*,* $[10, 12)$*,* $[12, 14)$*,* $[14, 16)$*, and* $[16, 17)$*.*



Each of the HI*-criticality jobs is decomposed into the sub-jobs shown in Figure 2; these are obtained by super-imposing the partitions shown above upon the EDF schedule constructed in Example 2.*

[4]Note that Step 1 may result in new break points to the time line and intervals other than release time and deadlines; e.g., $t = 6$.

J_i	a_i	c_i(HI)	d_i	χ_i
J_{12}	1	1.5	9	HI
J_{14}	1	0.5	12	HI
J_{15}	1	1	14	HI
J_{23}	9	0.5	10	HI
J_{24}	9	0.5	12	HI
J_{36}	10	0.5	16	HI
J_{37}	10	0.5	17	HI

Figure 2: HI-criticality sub-jobs generated by Step 3 in Example 1.

3.2 Run-time scheduling (EDF)

We maintain an EDF (priority) queue during run-time for a combination of the LO-criticality jobs and the HI-criticality sub-jobs (that were constructed during Step 3 above).

We now describe how run-time scheduling decisions are made during the ℓ'th interval I_ℓ, for $\ell = 1, 2, \ldots, k$:

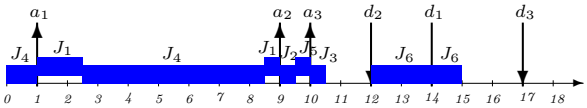
1. We first insert all LO-criticality jobs and HI-criticality sub-jobs that have their release time equal to the start of this interval, into the EDF queue.

2. We execute jobs (including sub-jobs) in EDF order, giving HI-criticality sub-jobs higher priority only when tie-breaking jobs with same deadlines.

   Note that from the manner in which the sub-jobs are defined, it is guaranteed that all HI-criticality sub-jobs with deadline at the end of this interval complete execution by the end of the interval, regardless of whether or when the processor degrades into slower speeds.

3. At the end of the interval, all jobs in the LO-criticality EDF queue with deadlines at the end of the interval are *dropped*. Dropping a job in this manner implies that LE-EDF was only able to schedule this instance in a partially correct manner (see Definition 1). We will later show (in the sense of online-optimal) that such drop is unavoidable, unless we have clairvoyance to speed changes.

EXAMPLE 4. *We continue scheduling the MC instance considered in Example 1 (jobs detailed in Figure 1; the* HI*-criticality sub-jobs constructed during Step 3 listed in Figure 2). The processor speed may fall below its nominal value at any instant during execution. To better illustrate how our algorithm works, we will separately simulate its operation under three different run-time behaviors of the processor.*
§1. *We first consider the case where no degradation in processor speed occurs, and all* HI*-criticality jobs execute at their* LO*-criticality WCETs. The schedule is depicted in the following figure. (Since sub-job numbers align with interval number, we only label the job numbers.)*



- *For Interval* $I_1 = [0, 1)$*, since no* HI*-criticality sub-job is allocated here,* J_4 *will be executed as the earliest deadline* LO*-criticality job.*

- *Sub-job J_{12} executes for 1.5 time units at the beginning of Interval $I_2 = [1, 9)$. The remaining capacity will be used for jobs with deadline greater than 9. As the earliest deadline* LO*-criticality job, J_4 executes first and completes at $t = 8.5$, after which J_{14} executes over the interval $[8.5, 9)$ (and also completes).*

- *Sub-job J_{23} is executed first in Interval $I_3 = [9, 10)$, and completes at time $t = 9.5$. The earliest deadline active job (which is J_5) executes over the interval $[9.5, 10)$.*

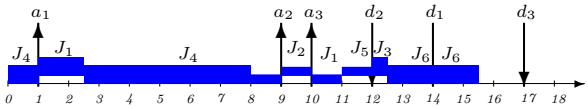
- *Since all* HI*-criticality jobs execute at their* LO*-criticality WCETs, both J_1 and J_2 are already finished at $t = 10$, and sub-jobs J_{15} and J_{24} require no execution. As a result,* HI*-criticality sub-job J_{36} (as the only active sub-job) will be executed in Interval $I_6 = [10, 10.5)$. We detect idleness throughout the rest of the interval; i.e., $[10.5, 12)$,*

- *Interval $I_5 = [12, 14)$ is empty and should be used for the only active job J_6.*

- *The only active* LO*-criticality job J_6 executes until it completes at $t = 15$. Now the processor becomes idle since J_{37} is an inactive sub-job, J_3 having already completed upon completing sub-job J_{36}.*

- *The processor idles during Interval $I_7 = [16, 17)$.*

§2. *Next we consider another case where the processor speed degrades to 0.5 over the time-interval $[8, 12)$. We assume that all* HI*-criticality jobs execute at their* LO*-criticality WCETs (and thus sub-jobs J_{15}, J_{24}, and J_{37} can be ignored[5]).*



- *Execution in Interval $I_1 = [0, 1)$ is the same as in previous case.*

- *Compared to the previous scenario, the amount of computing capacity available in Interval $I_2 = [1, 9)$ is less now due to the degradation of processor speed. After completing sub-job J_{12}, I_2 is only able to execute J_4, which completes at time-instant 9.*

- *Interval $I_3 = [9, 10)$ also suffers from the degradation, and is fully consumed by the sub-job J_{23}.*

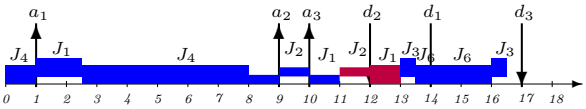
- *The processor remains in degraded mode for Interval $I_4 = [10, 12)$, where* HI*-criticality sub-job J_{14} executes and completes at time-instant 11. The remaining one time unit is used for executing* LO*-criticality job(s): J_5 executes from $t = 11$ to $t = 12$ and meets its deadline.*

- *The processor recovers to normal speed at time $t = 12$, and the executions in the remaining three intervals are the same as in the previous case.*

[5]Of course these sub-jobs will not actually be ignored during run-time; rather, they will be determined to be inactive (as it is explained in the case above). Here we simply ignore them in order to simplify the explanation.

Note that although the processor operated in degraded mode for four time units, LE-EDF nevertheless completed all the jobs by their deadlines.

§3. *As a final example, we consider the case where the processor suffers from a degradation between $t = 8$ and $t = 12$,* ***and*** HI*-criticality jobs J_1 and J_2 execute at their* HI*-criticality WCETs (for those reading this on a color monitor, execution beyond the* LO*-criticality WCET is depicted in purple).*



- *Execution in Intervals $I_1 = [0, 1)$, $I_2 = [1, 9)$, and $I_3 = [9, 10)$ remains the same as in the previous case.*

- *Both J_{14} and J_{24} need to complete within interval $I_4 = [10, 12)$. No capacity remains due to the processor degradation, and the unfinished* LO*-criticality job J_5 is dropped at its deadline $t = 12$.*

- *At the beginning of Interval $I_5 = [12, 14)$, the processor recovers to normal speed. The interval $[12, 13)$ is consumed by J_{15}. At time $t = 13$, there are two active jobs J_{36} and J_6 with the same deadline, and according to the algorithm, we favour* HI*-criticality jobs in such case, which results in execution of J_3 within $[13, 13.5)$, and then J_6 afterwards.*

- *There are two active jobs (J_{37} and J_6) within Interval $I_6 = [14, 16)$. J_6 executes first since it has got a earlier deadline (although with lower criticality level). Unfortunately, J_6 may be dropped at its deadline $t = 16$ since it has only received $2\frac{1}{2}$ units of execution (which is fewer than the required three units).*

- *Sub-job J_{37} executes in Interval $I_7 = [16, 17)$, completing at time-instant 16.5. The processor is idled for the remainder of the interval.*

■

It is instructive to review the last scenario considered in the example above, where two LO-criticality jobs J_5 and J_6 miss their deadlines.

The situation for J_5 within Interval $I_4 = [10, 12)$ is straightforward – the processor is suffering from a degradation within this interval, and since J_1 and J_2 are both HI-criticality jobs, the sub-jobs J_{14} and J_{24} certainly need to be prioritized over the LO-criticality job J_5.

The argument for J_6 to miss its deadline is not quite as unequivocal: a scheduling algorithm that postponed the execution of sub-job J_{36} to interval I_7 (where, as we saw, there is adequate excess capacity to accommodate this sub-job) and instead executed J_6 for an additional one-half unit during interval I_6 would have seen both J_6 and J_3 complete by their deadlines. However, such a scheduling algorithm would need to know beforehand (i.e., during interval I_6) that the processor speed would not degrade during interval I_7. that is, such an algorithm would need to be *clairvoyant* (see Section 2).

The failure of LE-EDF to correctly schedule an instance that would be scheduled correctly by a clairvoyant algorithm does not rule out the possibility that LE-EDF is an optimal

algorithm: according to Definition 2, an optimal scheduling strategy should be able to correctly schedule any instance that can be correctly scheduled by a non-clairvoyant scheduling strategy. Later in this paper we will prove (Theorem 1) that LE-EDF is optimal in the single WCET subcase (which is not NP-hard), and additionally seek to quantify the gap between clairvoyant and non-clairvoyant algorithms via the speedup factor metric.

Computational complexity. We have seen in Section 3.1 above that Algorithm LE-EDF generates no more than $\mathcal{O}(n)$ HI-criticality sub-jobs during the preprocessing phase; during run-time, these sub-jobs are scheduled for execution along-with the LO-criticality jobs. We note that standard techniques (see, e.g., [14]) for implementing EDF are known, that allow an EDF schedule for n jobs to be constructed in $\mathcal{O}(n \log n)$ time. Consequently, we conclude that the EDF-schedule of Step 2 can be constructed in $\mathcal{O}(n_{\text{HI}} \log n_{\text{HI}})$ time, and the total scheduler overhead during run-time is also bounded from above by $\mathcal{O}(n \log n)$.

Remark 1. The algorithm described in this section is triggered by, and quite similar to the one proposed in an earlier work (Section 4 of [10]). However, in that paper the authors only dealt with the uncertainty in platforms, but not with multi-WCET estimations alongside. The scheduling scheme in [10] was formed in a table driven manner; while in this paper, we manage to form it into a more implementation-friendly and efficient way (which is pure EDF during run-time upon job splitting). Moreover, further discussions and interesting comparisons (both theoretical and experimental) will be provided in the following sections.

Remark 2. LE-EDF applies for tasks with real number parameters – we restrict the examples with integer time only for easier demonstration and understanding. One may see that no such restriction is required in the correctness and optimality proof (in the following section).

4. THE *SINGLE WCET PER JOB* CASE

In this section, we consider MC instances in which each job has a single WCET specified for it, i.e., for each job J_i it is the case that $C_i(\text{LO}) = C_i(\text{HI})$. The scheduling of such MC instances was addressed in a recent paper [3], where an optimal scheduling strategy based upon linear programming (LP) was derived for scheduling such MC instances. We now prove, in Section 4.1 below, that LE-EDF is also an optimal scheduling strategy for scheduling such instances. Since (as we saw above) LE-EDF can be implemented to have a run-time that is $\mathcal{O}(n \log n)$ for an instance comprised of n jobs while LP-solvers have significantly poorer (although still polynomial) run-times, we argue that LE-EDF is a preferred algorithm for scheduling such instances.

In addition to proving the optimality of Algorithm LE-EDF for scheduling such MC instances, we use, in Section 4.2, the speedup factor metric to quantify the cost of clairvoyance by determining the smallest multiplicative factor by which the processor available to LE-EDF would need to be speeded up in order to be able to schedule any instance that can be scheduled by any (hypothetical) clairvoyant algorithm.

4.1 Proof of Optimality

LEMMA 1. *If a* LO*-criticality job J_i with release time a_i and deadline d_i is dropped by LE-EDF during run-time, the processor remains busy in interval $[a_i, d_i)$. Furthermore, no* HI*-criticality execution that had been allocated to later intervals (than d_i) in the pre-computed scheduling table gets executed within this interval.*

Proof: It is easy to see that job J_i remains *active* (released and unfinished) throughout this whole interval. Thus, there must be no idleness. Since our algorithm only "promotes" pre-allocated HI-criticality amounts when the processor idles, we know that no HI-criticality amount can be transferred from later intervals into $[a_i, d_i)$. ■

THEOREM 1. *LE-EDF is an* optimal scheduling strategy *for MC instances in which $C_i(\text{LO}) = C_i(\text{HI})$ for all jobs J_i.*

Proof: From the definition of an optimal scheduling strategy (Definition 2), it follows that we have two proof obligations here.

First, we must show that LE-EDF is able to schedule in a partially correct manner any instance that can be scheduled in a partially correct manner by any non-clairvoyant algorithm. Partial correctness trivially follows from the optimality of EDF: if any non-clairvoyant algorithm is able to satisfy property P2 of Definition 1, it follows from the manner in which we construct the scheduling table in Steps 1 and 2 of Section 3.1 that LE-EDF will also satisfy property P2.

Second, we must show that LE-EDF is able to correctly schedule any instance that can be correctly scheduled by any non-clairvoyant algorithm. Suppose that both LE-EDF and some other (non-clairvoyant) algorithm are both able to schedule a given MC instance $\mathcal{I}$ in a partially correct manner, but LE-EDF is unable to correctly schedule $\mathcal{I}$ – it drops a LO-criticality job J^* during run-time. Let a^* denote the release time, and d^* the deadline, of this job J^*. We argue that any non-clairvoyant scheduler that completes all HI-criticality jobs (and thereby satisfies partial correctness) must also fail to meet the deadline of J^* or some other LO-criticality job with deadline at or prior to time-instant d^*. This is because in order to ensure partial correctness in the event of the processor speed degrading to s_δ at some future point in time, a non-clairvoyant scheduler must make the most conservative assumptions regarding the future speed of the processor and assume that the speed will, indeed, fall to s_δ. But LE-EDF also makes this assumption, and ensures that under this assumption, the *minimum* possible amount of execution of HI-criticality jobs with deadline greater than d^* has occurred within the interval of interest. According to Lemma 1, no HI-criticality sub-job with deadline greater than d^* will be executed within $[a^*, d^*)$, since J^*, with an earlier deadline, is prioritized by LE-EDF. This implies that the maximum possible amount of execution to LO-criticality jobs has occurred in the LE-EDF schedule prior to d^*; the fact that LE-EDF is forced to nevertheless drop a job at d^* implies that the processor is overloaded prior to d^* (and hence no other algorithm can complete all LO-criticality jobs prior to d^*). ■

4.2 Speedup Factor

Theorem 1 above shows that LE-EDF is an optimal algorithm for scheduling MC instances in which each job's LO-criticality WCET is equal to its HI-criticality WCET, in the sense that no non-clairvoyant scheduler can guarantee correctness (partial correctness, respectively) if LE-EDF is unable to do so. Note that the proof of Theorem 1 fundamentally depends on the fact that the algorithm against

which LE-EDF is being compared is non-clairvoyant: a non-clairvoyant algorithm must necessarily assume at each instant during run-time that in the future the processor will execute throughout at its minimum (degraded) speed of s_δ. In contrast, a clairvoyant algorithm may know how the processor speed will vary in the future; such an algorithm will generally outperform LE-EDF since LE-EDF sometimes drops LO-criticality job to prevent future deadline missed by HI-criticality jobs due to possible processor degradation that may not happen. The third scenario considered in Example 4 had illustrated that a clairvoyant algorithm may ensure correctness while LE-EDF is only partially correct. In this section, we will quantify the gap between LE-EDF and any optimal clairvoyant algorithm using the metric of *speedup factor* [11]. The use of this metric for the purposes of quantifying the cost of non-clairvoyance seems particularly appropriate: the seminal paper [11] on speed factors was titled "*Speed is as powerful as clairvoyance*," which is what we, too, establish in this section (albeit for a completely different problem than the one considered in [11]).

In classical real-time scheduling theory (see, e.g.,[13, page 81]), the *load* of an instance of jobs denotes the maximum over all time intervals, of the cumulative execution requirement by jobs of the instance over the interval, normalized by the interval length. Informally, the load of an instance can be thought of as representing a lower bound on the speed of any processor upon which the instance can meet all deadlines. Analogous to this concept, we find it convenient to define two loads, $\ell_{\text{LO}}(\mathcal{J})$ and $\ell_{\text{HI}}(\mathcal{J})$, for any MC collection $\mathcal{J}$ of jobs.

DEFINITION 3. *The* LO*-criticality load* $\ell_{\text{LO}}(\mathcal{J})$ *and the* HI*-criticality load* $\ell_{\text{HI}}(\mathcal{J})$ *of a mixed-criticality collection* $\mathcal{J}$ *of jobs are defined according to the following two formulas:*

$$\ell_{\text{LO}}(\mathcal{J}) = \max_{0\le t_1<t_2} \frac{\sum_{J_i:\ t_1\le a_i\wedge d_i\le t_2} c_i(\text{LO})}{t_2-t_1};$$

$$\ell_{\text{HI}}(\mathcal{J}) = \max_{0\le t_1<t_2} \frac{\sum_{J_i:\ \chi_i=\text{HI}\wedge t_1\le a_i\wedge d_i\le t_2} c_i(\text{HI})}{t_2-t_1}.$$

It is easily seen that a necessary and sufficient condition for an optimal clairvoyant algorithm to successfully schedule MC instance $\mathcal{I} = (\mathcal{J}, s_\nu, s_\delta)$ is that $\ell_{\text{LO}}(\mathcal{J}) \le s_\nu$, and $\ell_{\text{HI}}(\mathcal{J}) \le s_\delta$. A natural question arises: can we determine a speedup factor $s(>1)$ for Algorithm LE-EDF such that a sufficient condition for LE-EDF to schedule MC instance $\mathcal{I} = (\mathcal{J}, s_\nu, s_\delta)$ in a correct manner (see Definition 1) is that $\ell_{\text{LO}}(\mathcal{J}) \le s\times s_\nu$, and $\ell_{\text{HI}}(\mathcal{J}) \le s\times s_\delta$? The following theorem leads us to an answer:

THEOREM 2. *If an MC instance* $\mathcal{I} = (\mathcal{J}, s\ell_{\text{LO}}(\mathcal{J}), s\ell_{\text{HI}}(\mathcal{J}))$ *that is schedulable by an optimal clairvoyant algorithm is not correctly scheduled by LE-EDF, then*

$$s < \frac{1}{1-\ell_{\text{HI}}(\mathcal{J})+\ell^2_{\text{HI}}(\mathcal{J})/\ell_{\text{LO}}(\mathcal{J})}. \tag{1}$$

Proof: (of Theorem 2). It is evident from the manner in which the scheduling table is constructed by Algorithm LE-EDF (in Steps 1–3) that a degraded speed of $\ell_{\text{HI}}(\mathcal{J})$ is already sufficient to have HI-criticality jobs meet their deadlines. It is straightforward to observe that LE-EDF is *sustainable* [2] with respect to processor speed (i.e., a faster processor would only reduce the execution time cost, and contribute positively its schedulability). Hence, LE-EDF remains correct if provided a faster processor which executes at degraded-speed of $s\ell_{\text{HI}}(\mathcal{J})$. As a result, if LE-EDF fails to maintain correctness for a given MC instance $\mathcal{I} = (\mathcal{J}, s\ell_{\text{LO}}(\mathcal{J}), s\ell_{\text{HI}}(\mathcal{J}))$, for any $s \ge 1$, the only possibility is that a LO-criticality job J_i is dropped at its deadline d_i – we study this only possible scenario in the following to derive a bound on the speedup factor s.

Based on Lemma 1, consider any interval $[a, d)$ which contains $[a_i, d_i)$; i.e., $a \le a_i$ and $d_i \le d$. Since we dropped a LO-criticality job at time $t = d_i$, the most pessimistic assumption is that our processor runs at degraded speed $s\ell_{\text{HI}}(\mathcal{J}_{\text{HI}})$ thereafter, and moreover we fully utilize interval $[d_i, d)$ with HI-criticality jobs. When compared to the clairvoyant execution of such a job set, the only difference for interval $[a, d_i)$ is that the additional capacity from the speedup $s(d_i - a_i)$ may be used to execute HI-criticality jobs with further deadlines. However, those HI-criticality jobs at the same time suffer from the degradation after time $t = d_i$, such that the provided capacity is not enough to guarantee them meeting deadlines. This is exactly the reason why our algorithm will pre-allocate more HI-criticality amounts into interval $[a, d_i)$, and thus cause the job J_i miss its deadline. Intuitively speaking, the additional capacity provided within interval $[a, d_i)$ is not enough to cover the "needs" from HI-criticality jobs that are executed later in interval $[d_i, d)$ by the clairvoyant algorithm. Thus the following inequality must hold for any $a \le a_i$, in order for LE-EDF to drop LO-criticality job J_i at its deadline.

$$\big(s\ell_{\text{LO}}(\mathcal{J})-\ell_{\text{LO}}(\mathcal{J})\big)(d_i-a) < \big(\ell_{\text{LO}}(\mathcal{J})-s\ell_{\text{HI}}(\mathcal{J})\big)(d-d_i) \tag{2}$$

The worst case is obtained by setting $a = a_i$, and this yields an upper bound on s. Without loss of generality, we assume $d - a_i = 1$, and denote $x := d - d_i \in [0, \ell_{\text{HI}}(\mathcal{J})]$. Since we only consider *active* HI-criticality jobs within the interval, x cannot exceed $\ell_{\text{HI}}(\mathcal{J})$ or else not even a clairvoyant algorithm would finish them on time. Inequality (2) can be re-written in the following manner with respect to the speedup factor s:

$$\forall x \in [0, \ell_{\text{HI}}(\mathcal{J})], s < \frac{1}{1-x+x\frac{\ell_{\text{HI}}(\mathcal{J})}{\ell_{\text{LO}}(\mathcal{J})}} \tag{3}$$

When $\ell_{\text{HI}}(\mathcal{J}) \ge \ell_{\text{LO}}(\mathcal{J})$, we simply have $s < 1$ which is not the case of interest. When $\ell_{\text{HI}}(\mathcal{J}) < \ell_{\text{LO}}(\mathcal{J})$, the right hand side of 3 is monotonically increasing with respect to x, the upper bound of the speedup factor becomes tight when x takes its largest possible value $\ell_{\text{HI}}(\mathcal{J})$, which will lead us to:

$$s < \frac{1}{1-\ell_{\text{HI}}(\mathcal{J})+\ell^2_{\text{HI}}(\mathcal{J})/\ell_{\text{LO}}(\mathcal{J})}.$$

and the theorem follows. ■

Analysis of Inequality (1) yields the following corollary.

COROLLARY 1. *The upper bound of the speedup factor is* $s_{\max} = 4/3$*, which occurs when* $\ell_{\text{LO}}(\mathcal{J}) = 1$ *and* $\ell_{\text{HI}}(\mathcal{J}) = 0.5$. ■

5. THE *CONSTANT-SPEED* CASE

As stated in Section 1 above, most prior work in MC scheduling has focused upon a workload model in which multiple WCET estimates are provided for each job while the processor speed is assumed to be bounded from below throughout runtime by an *a priori* known constant. An algorithm named OCBP (for **O**wn **C**riticality **B**ased **P**riorities) was proposed in [4] for scheduling such instances, and shown to have a speedup bound of $(\sqrt{5}+1)/2$ (i.e., ≈ 1.62); to date, this is the best speedup bound known for any algorithm for scheduling such MC instances. We will now show that in scheduling MC instances implemented upon processors whose speed bound does not vary during run-time, LE-EDF strictly dominates OCBP in the sense of correctly scheduling all instances that are correctly scheduled by OCBP as well as some additional ones that are not scheduled correctly by OCBP. (It follows from this domination relation that the speedup bound for LE-EDF is also at least $(\sqrt{5}+1)/2$, upon such processors with constant speed bound.)

We start out briefly describing OCBP. Given an MC instance $\mathcal{J}$, OCBP derives off-line a priority ordering for all jobs in the instance, using a variant of the Audsley Optimal Priority Assignment (OPA) scheme [1], in the following manner (here, "scheduling according to priority" means that at each moment in time the highest-priority available job is executed). It determines, as described below, the job that may be assigned lowest priority, and assigns it the lowest priority. This procedure is repeated upon the set of jobs excluding the lowest priority job, until all jobs are ordered, or at certain iteration a lowest priority job cannot be found.

1. We assign lowest priority to the latest-deadline LO-criticality job if it would complete by its deadline when every other job were assigned higher priority and execute in their LO-criticality level WCETs.

2. Else, we assign lowest priority to the latest-deadline HI-criticality job if it would complete by its deadline when every other job were assigned higher priority and execute in their HI-criticality level WCETs. Here LO-criticality jobs' HI-criticality level WCETs remains the same as their LO-criticality level WCETs.

3. Else, we declare failure.

The following theorem asserts that any instance that can be scheduled by OCBP is also scheduled by LE-EDF.

THEOREM 3. *Given any set of MC jobs $\mathcal{J}$, if Algorithm LE-EDF fails to complete job(s) at some criticality level on time (either by missing a deadline, or dropping a job), then so will OCBP.*

Proof: There are only two steps during execution at which Algorithm LE-EDF may report a failure to correctly schedule an instance.

If Algorithm LE-EDF fails at Step 1 when constructing schedule table for HI-criticality jobs, it directly follows that there is no *correct* schedule scheme for HI-criticality jobs when they all execute at their HI WCETs. Thus OCBP algorithm will also fail to correctly schedule this instance.

Now we consider the case that Algorithm LE-EDF fails during run-time, which indicates that some LO-criticality job J_i missed its deadline and will be dropped at time $t = d_i$. We will show that OCBP algorithm must also drop a LO-criticality job at or before this time in order to guarantee correctness of HI-criticality job execution.

One subcase is that a HI-criticality job has executed longer than its LO-criticality WCET before time $t = d_i$. OCBP algorithm will immediately drop remaining LO-criticality jobs when it occurs, which is at or before time $t = d_i$.

The other subcase is that all HI-criticality jobs have executed no longer than their LO-criticality WCETs (so far). We will prove by contradiction that OCBP algorithm will also drop some job at or before time $t = d_i$.

Assume OCBP algorithm has not dropped any job at or before time $t = d_i$, which means that it makes so far all jobs meet their deadlines at time $t = d_i$. Consider only the LO-criticality jobs, from the description we can easily tell that both algorithms execute them at an EDF order: OCBP generates the priority list by considering jobs in each criticality level in non-increasing order of deadlines, while LE-EDF uses the remaining capacity for all LO-criticality jobs in the EDF order (after pre-allocating and slicing HI-criticality jobs). Since LE-EDF fails to meet some LO-criticality job's deadline at d_i while OCBP does not, it must be the case that OCBP executes LO-criticality jobs (totally) between time $t = 0$ and $t = d_i$ for a longer time than LE-EDF. Thus OCBP has executed less amounts of HI-criticality jobs until time $t = d_i$ than LE-EDF [6]. However LE-EDF guarantees that at this deadline d_i, all HI-criticality sub-jobs with deadline on or before d_i are "must to be finished", which means that a shorter accumulated execution time to HI-criticality jobs will cause a deadline miss in the future. This contradicts the correctness guarantee to HI-criticality jobs of OCBP, and indicates that our assumption that OCBP algorithm have not dropped any job at or before time $t = d_i$ is incorrect. The theorem results from this contradiction. ■

The following example illustrates that there are instances correctly scheduled by Algorithm LE-EDF, that OCBP fails to schedule correctly. This example, in conjunction with Theorem 3 above, allows us to conclude that Algorithm LE-EDF *dominates* OCBP.

EXAMPLE 5. *Consider the instance consisting of the following six jobs shown in tabular form in Figure 3 below, to be scheduled on a unit-speed processor. This instance is not*

J_i	a_i	c_i(LO)	c_i(HI)	d_i	χ_i
J_1	1	2	4	14	HI
J_2	9	1	2	12	HI
J_3	10	1	2	16	HI
J_4	0	8	8	10	LO
J_5	1	1	1	12	LO
J_6	12	3	3	16	LO

Figure 3: The MC instance considered in Example 5.

OCBP-schedulable: it may be validated that after assigning J_6 the lowest priority, no job can be further assigned the second lowest priority.

We now show that LE-EDF schedules this instance correctly. Prior to run-time Algorithm LE-EDF identifies the intervals during which reservations are made in the scheduling table, and applies EDF to HI-criticality jobs assuming

[6]Both algorithms have exactly the same idleness periods since (by definition) both will idle the processor only when there is no *active* job.

(Please view this figure upon a color monitor/ printout.)

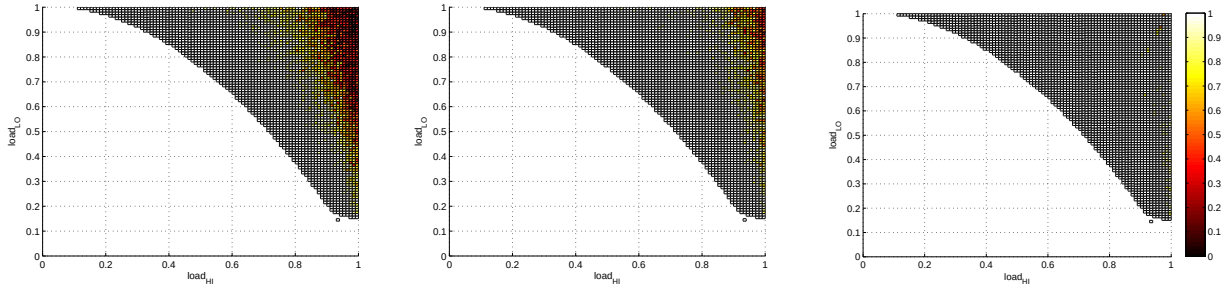
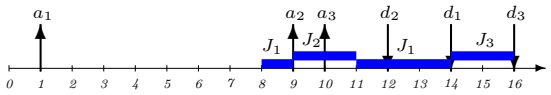
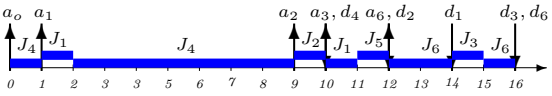


Figure 4: Schedulability comparison of OCBP (left), MCEDF (middle), and LE-EDF (right), where the color of each block represents the fraction of schedulable instances with ℓ_{LO} and ℓ_{HI} parameters falling within certain small ranges. (Informally, red is better – observe that there is quite a bit of blue in the upper right segment of the plot for OCBP, less for MCEDF, and almost none for LE-EDF.)

they all execute for HI*-criticality WCETs. The constructed scheduling table looks as follows:*



If all HI*-criticality jobs execute for no more than their* LO*-criticality WCETs during run-time, then the remaining capacity is enough for LE-EDF to accommodate all the* LO*-criticality jobs to meet their deadlines; this is illustrated in the schedule below:*



Comparison with MCEDF [16]

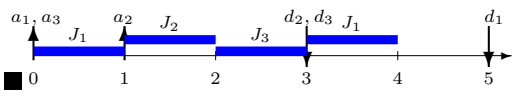
An algorithm named MCEDF was recently [16] presented for scheduling MC instances upon processors that are speed bounded by a constant during run-time –i.e., the same kind of workload scheduled by OCBP– and shown to strictly dominate OCBP (to the best of our knowledge, MCEDF is the only algorithm proven to dominate OCBP). We do not yet know whether LE-EDF dominates MCEDF or not; we do, however, show below that the converse cannot be the case.

THEOREM 4. *There are MC instances correctly scheduled by Algorithm LE-EDF that MCEDF does not schedule in a correct manner.*

Proof: We present one such instance:

J_i	a_i	c_i(LO)	c_i(HI)	d_i	χ_i
J_1	0	2	3	5	HI
J_2	1	1	2	3	HI
J_3	0	1	1	3	LO

It was shown in [16] that this instance is not MCEDF schedulable. The following schedule show how LE-EDF schedules this instance, and correctness is thus verified from this schedule.



Simulations

We performed some simulation experiments to complement the theoretical conclusions of Theorems 3 and 4. The experimental setup is as described in [16, Sec. IV] (we are grateful to the authors of [16], Dario Socci in particular, for sharing the source-code of MCEDF and their workload-generator, and for assistance in conducting our experiments). As in [16], we generate a large number of MC instances of 20 jobs each. The parameters $\ell_{\text{LO}}(\cdot)$ and $\ell_{\text{HI}}(\cdot)$ of the generated instances range from 0 to 1, with step 0.01. Only "overloaded" instances –those satisfying $\ell^2_{\text{LO}}(\mathcal{J}) + \ell_{\text{HI}}(\mathcal{J}) > 1$– are considered since all three algorithms are successful in scheduling the non-overloaded ones. Among the 33511 successfully generated instances, OCBP fails to schedule 5076 ($\approx 15.1\%$). From amongst these[7], MCEDF reports failure as well for 1986 ($\approx 5.9\%$), only 109 ($\approx 0.3\%$) of which are also unschedulable by LE-EDF. Further, all the instances scheduled by MCEDF (and OCBP) were also scheduled by LE-EDF. Figure 4 depicts the schedulability results for the three algorithms. Instances with similar ℓ_{LO} and ℓ_{HI} values are put into a same small block (with typical size of 10 to 15 instances). The color of each small block represents the percentage of schedulable sets.

In all these and several other experiments not discussed here, we have not been able to identify any instance that can be scheduled by MCEDF but not by LE-EDF. Although this certainly does not constitute formal proof that LE-EDF dominates MCEDF, it seems clear that generally speaking, LE-EDF is superior to the other two existing algorithms, both in terms of schedulability (as shown in the experiments), and run-time complexity (theoretically shown to be $\mathcal{O}(n \log n)$, where $n = |\mathcal{J}|$, which is asymptotically much better than OCBP and MCEDF's $\mathcal{O}(n^2 \log n)$).

6. CONTEXT & CONCLUSIONS

Scheduling theory is applied to the analysis of *models* of systems, rather than to the physical systems themselves. In order to have confidence that the conclusions drawn on the basis of the analysis of such models will hold for the actual systems being modeled, the modeling process typi-

[7]There are no instances scheduled by OCBP but not MCEDF – this is as expected, since MCEDF was shown [16] to dominate OCBP.

cally incorporates considerable pessimism into the model; such pessimism gets reflected during run-time in the form of under-utilization of platform resources that were provisioned on the basis of the pessimistic models.

Mixed-criticality (MC) scheduling theory seeks to deal with such pessimism by constructing multiple different models of a single system, and using more pessimistic models for validating the correctness of more critical functionalities whose correctness must be validated to a higher level of assurance. Prior work in MC scheduling has separately dealt with modeling pessimism along different dimensions, including estimating *upper bounds on the WCET* of pieces of code, and estimating *lower bounds on processor speed* during run-time. In this paper, we have considered both these dimensions within a single integrated framework.

We have proposed a formal model for representing such mixed-criticality systems that are comprised of a finite collection of independent jobs executing upon a single preemptive processor, and have derived an algorithm, LE-EDF, for scheduling such instances. We have shown that LE-EDF can be implemented in an efficient manner to have run-time complexity that is a low-order polynomial in the representation of the MC instance being scheduled. We have quantified the schedulability gap between LE-EDF and any hypothetical clairvoyant algorithm by determining a speedup bound for LE-EDF. We have also shown that LE-EDF is a strict improvement over some previously-proposed algorithms that deal with only one of these two dimensions of pessimism:

- With regards to pessimism only with respect to processor speed, LE-EDF retains the optimality property of a previously-proposed algorithm [3], while having a more efficient implementation: while the algorithm in [3] requires the solution of an LP, LE-EDF is based on EDF scheduling.
- With regards to pessimism only with respect to estimating the WCET, LE-EDF strictly dominates the OCBP scheduling algorithm [4]. It is also able to schedule some instances that are known to not be schedulable by MCEDF, another recently-proposed [16] algorithm.

The research reported in this document can be extended in several directions. An obvious and important extension is to MC systems characterized by *more than just two criticality levels*. Although the multiple-WCET model has been successfully extended in this manner, one encounters some interesting challenges in attempting to generalize results concerning varying-speed processors to incorporate more than two criticality levels. We are currently working on addressing these challenges; once we have succeeded, we will seek to integrate these results with the pre-existing ones for handling multiple WCETs, thereby once again enabling the concurrent modeling of uncertainty in estimating both WCET and processor speed.

In this paper, we have considered real-time workloads that are specified as finite collections of *jobs*, whereas collections of *recurrent tasks* is typically of more interest for developers of real-time systems. Results concerning the MC scheduling of recurrent tasks have previously been obtained that deal solely with pessimism regarding estimating the WCET. We have recently obtained some results for dealing with pessimism in estimating processor speed when scheduling recurrent task systems, and are working on unifying the results along both dimensions into an integrated framework – such unification appears to be non-trivial.

7. REFERENCES

[1] N. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[2] S. Baruah and A. Burns. Sustainable scheduling analysis. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 159–168, 2006.

[3] S. Baruah and Z. Guo. Mixed-criticality scheduling upon varying-speed processors. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, RTSS 2013.

[4] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2010.

[5] D. Bull, et al. A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient- error tolerance and adaptation to PVT variation. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 284–285, 2010.

[6] A. Burns and R. Davis. Mixed-criticality systems: A review. 2013. Available at `http://www-users.cs.york.ac.uk/~burns/review.pdf`.

[7] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Second edition, 2005.

[8] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[9] Z. Guo and S. Baruah. Mixed-criticality scheduling upon unmonitored unreliable processors. In *Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*, 2013.

[10] Z. Guo and S. Baruah. Implementing mixed-criticality systems upon a preemptive varying-speed processor. *Leibniz Transactions on Embedded Systems (LITES)*, 1(2):3:1–3:19, 2014.

[11] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 37(4):617–643, 2000.

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

[13] J. Liu. *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 2000.

[14] A. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, 1988.

[15] D. Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, RTSS 2009, 2009.

[16] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. In *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, 2013.

[17] R. Wilhelm, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008.