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

Zhishan Guo
The University of North Carolina at Chapel Hill
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 reinterpret 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-
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 `cupspeed` 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 \(c_i(lo), c_i(\text{hi})\) of two WCET values where \(c_i(lo) \leq c_i(\text{hi})\) for II-criticality jobs and \(c_i(lo) = c_i(\text{hi})\) for LO-criticality ones, a deadline \(d_i\), and a criticality level\(^1\) \(\chi_i \in \{1, 2\}\).

A mixed-criticality instance \(I\) is specified by

- a collection of MC jobs: \(J = \{J_1, J_2, \ldots, J_n\}\), and
- a processor that is characterized by two thresholds: a normal speed \(s_n\) and a degraded speed \(s_d(\leq s_n)\).

The interpretation is that the jobs in \(J\) 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-\(s_n\) (or faster) processor and hence completes at least \(s_n\) units of execution per unit time, whereas in degraded mode it completes less than \(s_n\), but at least \(s_d\) units of execution per unit time. 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 \(I = (\{J_1, J_2, \ldots, J_n\}, s_n, s_d)\), 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 \(\geq s_n\) throughout Interval \([a_i, d_i]\); and

**P2.** Each III-criticality job \(J_i\) meets its deadline if all III-criticality jobs complete execution upon having executed for no more than their LO-criticality WCETs, and the processor speed remains \(\geq s_d\) 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 III-criticality jobs provided the processor executes at or above its degraded speed and each II-criticality job completes execution for no more than its LO-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 $I$, then no non-clairvoyant algorithm can ensure correctness (partial correctness, resp.) for the instance $I$.

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 $I$ are indexed $1, 2, \ldots, n_h$, and the lo-criticality jobs are indexed $n_h+1, \ldots, n$. Let $I_1, I_2, \ldots, I_k$ 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 $V_j, t_j < t_{j+1}$). These release time and deadlines partition the time duration $[\min\{a_i\}, \max\{d_i\}]$ into $k$ intervals, which will be denoted as $I_1, I_2, \ldots, I_k$, 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 $I = (J, s_c, s_\delta)$, 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 schedulable 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-$s_d$ processor, if

1. each job executes for its hi-criticality WCET,
2. each job completes by its deadline, and

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. 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 $O(n_h \log n_h)$ time (which is the complexity of sorting), where $n_h$ 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 $s_c = 1$ and degraded speed $s_d = 0.5$.

<table>
<thead>
<tr>
<th>$J_i$</th>
<th>$a_i$</th>
<th>$c_i(\text{LO})$</th>
<th>$c_i(\text{HI})$</th>
<th>$d_i$</th>
<th>$\chi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$J_2$</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>$J_3$</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$J_4$</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 1: An example MC collection of jobs.

Considering only the hi-criticality jobs $J_1$–$J_6$ executing for their hi-criticality WCETs on a speed-$s_d$ processor, the intervals identified in Step 1 are as follows:

The intervals determined in Step 1 are therefore $[0, 14]$ 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 $s_d$ 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

\(^2\)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 \( \text{HI} \)-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 \([\min(a_i), \max(d_i)]\), 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 \( \text{LO} \)– \( \text{LO} \)-criticality and \( \text{HI} \)-criticality.) For each \( \text{HI} \)-criticality job \( J_i \) and each interval \( I_k \) 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 \( I_k \), and a deadline equal to \( t_{k+1} \), the right end-point of Interval \( I_k \).

By dividing \( \text{HI} \)-criticality jobs into sub-jobs, and setting proper deadlines for them (so that they cannot be suppressed by \( \text{LO} \)-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 \( 3n_k - 1 \) contiguous chunks of execution (here, a \( 2n_k - 1 \) comes from the fact that \( n_k \) jobs are being scheduled using EDF, and an additional \( n_k \) from the fact that there may be as many as \( n_k \) 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 \( 3n_k - 1 + 2n - 1 \), which is \( O(n) \).

**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 \( \text{HI} \)-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., \( t = 6 \).

<table>
<thead>
<tr>
<th>( J_i )</th>
<th>( a_i )</th>
<th>( c_i(\text{HI}) )</th>
<th>( d_i )</th>
<th>( x_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( J_{12} )</td>
<td>1</td>
<td>1.5</td>
<td>9</td>
<td>( \text{HI} )</td>
</tr>
<tr>
<td>( J_{14} )</td>
<td>1</td>
<td>0.5</td>
<td>12</td>
<td>( \text{HI} )</td>
</tr>
<tr>
<td>( J_{15} )</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>( \text{HI} )</td>
</tr>
<tr>
<td>( J_{23} )</td>
<td>9</td>
<td>0.5</td>
<td>10</td>
<td>( \text{HI} )</td>
</tr>
<tr>
<td>( J_{24} )</td>
<td>9</td>
<td>0.5</td>
<td>12</td>
<td>( \text{HI} )</td>
</tr>
<tr>
<td>( J_{36} )</td>
<td>10</td>
<td>0.5</td>
<td>16</td>
<td>( \text{HI} )</td>
</tr>
<tr>
<td>( J_{57} )</td>
<td>10</td>
<td>0.5</td>
<td>17</td>
<td>( \text{HI} )</td>
</tr>
</tbody>
</table>

**Figure 2:** \( \text{HI} \)-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 \( \text{LO} \)-criticality jobs and the \( \text{HI} \)-criticality sub-jobs (that were constructed during Step 3 above).

We now describe how run-time scheduling decisions are made during the \( t \)-th interval \( I_t \), for \( t = 1, 2, \ldots, k \):

1. We first insert all \( \text{LO} \)-criticality jobs and \( \text{HI} \)-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 \( \text{HI} \)-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 \( \text{HI} \)-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 \( \text{LO} \)-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 \( \text{HI} \)-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.

**S1.** We first consider the case where no degradation in processor speed occurs, and all \( \text{HI} \)-criticality jobs execute at their \( \text{LO} \)-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 \( \text{HI} \)-criticality sub-job is allocated here, \( J_4 \) will be executed as the earliest deadline \( \text{LO} \)-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 $[8, 9, 10]$ (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 $J_{30}$ (as the only active sub-job) will be executed in Interval $I_0 = [10, 10.5)$. We detect idleness throughout the rest of the interval; i.e., $[10.5, 12)$.

Interval $I_1 = [12, 14)$ 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 $I_7 = [16, 17)$.

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

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 $I_2 = [10, 12)$. No capacity remains due to the processor degradation, and the unfinished lo-criticality job $J_3$ 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 $J_{26}$ 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 $J_7$ 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 \( O(n) \) 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 \( O(n \log n) \) time. Consequently, we conclude that the EDF-schedule of Step 2 can be constructed in \( O(n \log n) \) time, and the total scheduler overhead during run-time is also bounded from above by \( O(n \log n) \).

Remark 1. The algorithm described in this section is triggered by, and quite similar to the one proposed in an earlier paper, we manage to form it into a more implementation-friendly and efficient way (which is pure EDF during run-time that is processed by, and quite similar to the one proposed in an earlier paper, we manage to form it into a more implementation-friendly and efficient way (which is pure EDF during run-time). In this paper, we form to manage it into a more implementation-friendly and efficient way (which is pure EDF during run-time). In this paper, we manage to form it into a more implementation-friendly and efficient way (which is pure EDF during run-time).

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 \( J \), it is the case that \( C_i(lo) = C_i(hi) \). 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 \( O(n \log n) \) 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 \( J \), with release time \( a_r \) and deadline \( d_r \), is dropped by LE-EDF during run-time, the processor remains busy in interval \([a_r, d_r]\). Furthermore, no hi-criticality execution that had been allocated to later intervals (than \( d_r \)) in the pre-computed scheduling table gets executed within this interval.

Proof: It is easy to see that job \( J \) 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 \([a_r, d_r]\).

Theorem 1. LE-EDF is an optimal scheduling strategy for MC instances in which \( C_i(lo) = C_i(hi) \) 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 \( I \) in a partially correct manner, but LE-EDF is unable to correctly schedule \( I \) – 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 the processor speed degrading to \( s_2 \) 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 \( s_2 \). 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 \( J^* \) has an earlier deadline, is prioritized by LE-EDF. This implies that the maximum possible amount of execution of 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 \( d^* \)).
which LE-EDF is being compared is non-clairvoyant: a non-
clairvoyant algorithm must necessarily assume at each in-
nstant during run-time that in the future the processor will
execute throughout at its minimum (degraded) speed of
$s_d$. 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 Ex-
ample 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 par-
ticularly 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 com-
pletely 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 require-
ment 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 dead-
lines. Analogous to this concept, we find it convenient to
define two loads, $\ell_{LO}(J)$ and $\ell_{HI}(J)$, for any MC collection
$J$ of jobs.

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

$$
\ell_{LO}(J) = \max_{0 \leq t_1 < t_2} \sum_{j_i : t_2 \leq m_i < d_i \leq t_2} c_{i}(LO) = \frac{t_2 - t_1}{t_2 - t_1};
$$

$$
\ell_{HI}(J) = \max_{0 \leq t_1 < t_2} \sum_{j_i : t_1 < m_i < d_i \leq t_2} c_{i}(HI) = \frac{t_2 - t_1}{t_2 - t_1}.
$$

It is easily seen that a necessary and sufficient condition
for an optimal clairvoyant algorithm to successfully schedu-
le MC instance $I = (J, s, s_d)$ is that $\ell_{LO}(J) \leq s_0$, and
$\ell_{HI}(J) \leq s_2$. 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
$I = (J, s, s_d)$ in a correct manner (see Definition 1) is that
$\ell_{LO}(J) \leq s \times s_0$, and $\ell_{HI}(J) \leq s \times s_2$? The following theorem
leads us to an answer:

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

$$
s < \frac{1}{1 - \ell_{HI}(J)/\ell_{LO}(J)}.
$$

**Proof.** (of Theorem 2). It is evident from the man-
nner in which the scheduling table is constructed by Al-
gorithm LE-EDF (in Steps 1–3) that a degraded speed of
$\ell_{HI}(J)$ 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 $s_{HI}(J)$. As a result, if
LE-EDF fails to maintain correctness for a given MC in-
stance $I = (J, s_{LO}(J), s_{HI}(J))$, for any $s \geq 1$, 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 con-
ains $[a_i, d_i]$: i.e., $a \leq a_i$, and $d_i \leq d$. Since we dropped
a Lo-critical job at time $t = d_i$, the most pessimistic
assumption is that our processor runs at degraded speed
$s_{HI}(J_{HI})$ thereafter, and moreover we fully utilize inter-
val $[d_i, d]$ with hi-criticality jobs. When compared to the
clairvoyant execution of such a job set, the only difference
for interval $[a, d]$ 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-critical jobs at
the same time suffer from the degradation after time $t = d_i$,
such that the provided capacity is not enough to guaran-
tee 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 in-
equality must hold for any $a \leq a_i$, in order for LE-EDF to
drop Lo-criticality job $J_i$ at its deadline.

$$
(s_{LO}(J_i) - \ell_{LO}(J_i))(d_i - a) < (\ell_{HI}(J_i) - s_{HI}(J_{HI}))(d_i - d_i).
$$

The worst case is obtained by setting $a = a_i$, and this yields
an upper bound on $s$. Without loss of generality, we assume
$d_i - a_i = 1$, and denote $x := d_i - d_i \in [0, \ell_{HI}(J_i)]$. Since we only
consider active hi-criticality jobs within the interval, $x$
cannot exceed $\ell_{HI}(J)$ 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
$s$:

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

When $\ell_{HI}(J) \geq \ell_{LO}(J)$, we simply have $s < 1$ which is
not the case of interest. When $\ell_{HI}(J) < \ell_{LO}(J)$, 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 $\ell_{HI}(J)$, which will lead us to:

$$
s < \frac{1}{1 - \ell_{HI}(J)/\ell_{LO}(J)}.
$$

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 $\ell_{LO}(J) = 1$ and $\ell_{HI}(J) = 0.5$. □
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**n **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., $\approx 1.62$); 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 correctness of LO-criticality WCETs. Prior to run-time Algorithm LE-EDF identifies the priority ordering for all 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. 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 LO-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 OCBP-schedulable: it may be validated that after assigning $J_6$, the lowest priority, no job can be further assigned the second lowest priority.

<table>
<thead>
<tr>
<th>$J_i$</th>
<th>$a_i$</th>
<th>$c_i \text{(LO)}$</th>
<th>$c_i \text{(HI)}$</th>
<th>$d_i$</th>
<th>$\chi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td>HI</td>
</tr>
<tr>
<td>$J_2$</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>HI</td>
</tr>
<tr>
<td>$J_3$</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>HI</td>
</tr>
<tr>
<td>$J_4$</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>LO</td>
</tr>
<tr>
<td>$J_5$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>LO</td>
</tr>
<tr>
<td>$J_6$</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>LO</td>
</tr>
</tbody>
</table>

**Figure 3:** The MC instance considered in Example 5.
they all execute for \( \ell_i \)-criticality WCETs. The constructed scheduling table looks as follows:

\[
\begin{array}{ccccccc}
\ell_1 & a_1 & c_1 & \ell_2 & a_2 & d_1 & d_2 \\
\ell_3 & 1 & 1 & \ell_4 & J_3 & & \\
& & & & & & \\
\end{array}
\]

\( 0, 1, 2, 3, 4, 5 \)

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

![Scheduling Table](image)

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:

\[
\begin{array}{ccccccc}
J_1 & a_1 & c_1 & d_1 & J_2 \\
& 0 & 2 & 3 & 5 & 3 & \chi \\
J_3 & 1 & 1 & 3 & 3 & 3 \\
& 0 & 1 & 1 & \ell_o, d_2, d_3 \\
\end{array}
\]

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.

![Scheduling Schedule](image)

**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 \( \ell_{\ell_i}(\cdot) \) and \( \ell_{\ell_i}(\cdot) \) of the generated instances range from 0 to 1, with step 0.01. Only “overloaded” instances –those satisfying \( \ell_{\ell_i}(J) + \ell_{\ell_i}(J) > 1 \) – 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 (≈ 15.1%). From amongst these\(^7\), MCEDF reports failure as well for 1986 (≈ 5.9%), only 109 (≈ 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 \( \ell_i \) 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 \( O(n \log n) \), where \( n = |J| \), which is asymptotically much better than OCBP and MCEDF’s \( O(n^2 \log n) \)).

---

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