Semi-Clairvoyance in Mixed-Criticality Scheduling

Kunal Agrawal*, Sanjoy Baruah*, Alan Burns†
*Washington University in St. Louis
†University of York

***Abstract*—In the Vestal model of mixed-criticality systems, jobs are characterized by multiple different estimates of their actual, but unknown, worst-case execution time (WCET) parameters. Prior work on mixed-criticality scheduling theory assumes that the execution duration of a job is only revealed by actually executing the job through to completion. We consider a different *semi-clairvoyant* model here, in which it is assumed that upon arrival a job reveals which of its WCET parameters it will respect. We identify circumstances under which this is a reasonable model, and design and evaluate scheduling algorithms appropriate for this model. We show that such semi-clairvoyance yields a significant quantifiable benefit over non-clairvoyance, in terms of both the complexity of schedulability analysis and the speedup needed to ensure schedulability.**

I. Introduction

A model for *mixed-criticality* workloads was proposed by Vestal [1] as a means of achieving timing predictability upon modern processors. The worst-case execution time (WCET) parameters of pieces of code — the maximum duration of time that a piece of code may take to complete execution — play a crucial role in ensuring timing predictability. However, the inherent unpredictability, variation, and uncertainty in the execution duration of pieces of code upon modern processors means that the true WCET of a piece of code is both very difficult to determine exactly, and of relatively limited use since it can be several orders of magnitude greater than the execution duration under the vast majority of circumstances — it typically takes a pathological and extremely unlikely combination of run-time conditions (a "perfect storm") for the true worst-case timing behavior to be realized in practice. In the Vestal model, individual pieces of real-time code are modeled as schedulable entities called *jobs* that have deadlines associated with them, and that are characterized by multiple WCET parameters.[1] These different WCET parameters represent different estimates, made at differing levels of assurance, of the "true" (unknown) WCET of the code. Each job is also assigned a criticality — in the two-criticality level model that we will be considering in this paper, these are called HI and LO, denoting greater and lesser criticality respectively, and the two WCET parameters, one at a level of assurance consistent with HI criticality and a second at a level of assurance consistent with LO criticality, are determined for each job. The correctness criterion in the Vestal mixed-criticality model is that if all the jobs complete execution within a duration not exceeding their LO-criticality WCET estimates then all the jobs should complete execution by their respective deadlines, while if some jobs do not complete execution within their LO-criticality WCET estimates (but all jobs would complete execution if allowed to execute for as much as their HI-criticality WCET), then at least the HI-criticality jobs should complete execution by their respective deadlines.

Despite some very reasonable questions regarding its scope and applicability [2], [3] that has given rise to an interesting and intellectually engaging debate (see, e.g., [4], [5]), the Vestal model has attracted a lot of attention in the real-time scheduling theory community, and has engendered a large body of research that explores various aspects of real-time scheduling under the assumptions of this model (see, [6] for a reasonably current and very comprehensive survey). In most of this research it is assumed that the only way to find out whether a job will complete execution within its LO-criticality WCET is to actually execute it until it has either completed, or has received an amount of execution equal to its LO-criticality WCET without completing. In this paper, we consider a somewhat different model: we assume that it becomes known when a job *arrives* (i.e., it is released for execution) whether it will complete execution within a duration $\leq$ its LO-criticality WCET or not. There are several kinds of mixed-criticality systems for which such an assumption makes sense from a pragmatic perspective. First, the system developer may provide *alternative implementations* of a job: upon arrival, the job knows which implementation is the correct one to execute under the current circumstances. (E.g., there may be an implementation for execution under regular conditions, and another for execution under unexpected – HI-criticality – conditions: presumably the HI-criticality implementation incorporates crisis-mitigation functionalities for dealing with the critical conditions.). Additionally, the execution time of a piece of code typically depends upon the state of the system at the time the code is executing; since the system state is better known during run-time when a job arrives (than during job-design time), one can presumably obtain a more accurate estimation of the actual duration of execution for the just-arrived job.[2]

Semi-clairvoyance. In the context of mixed-criticality workloads, we will refer to the property of knowing, upon a job's arrival, whether it will complete execution within a duration not exceeding its LO-criticality WCET estimate as *semi-clairvoyance*. This terminology is derived from the previously-

[1] In this paper as in much of the current research on mixed-criticality scheduling, we restrict our attention to two WCET estimates per job.

[2] This is particularly true if *parametric* WCET analysis techniques [7], [8] had previously been used to obtain parametrized expressions for the WCET, the values of these parameters being determined at run-time.

introduced concept of *clairvoyant* schedulers [9], which are assumed to know beforehand the precise actual duration for which each job will execute. While clairvoyant schedulers are an idealized abstraction that are unlikely to exist for any but the most deterministic actual systems, semi-clairvoyant schedulers are, as we have argued above, realizable for a variety of kinds of workloads. In addition to its potential practical applicability, we believe that the study of semi-clairvoyance is interesting from a theoretical perspective since it helps provide a better understanding as to the benefits of fore-sight: it provides an additional data-point for exploring the tradeoff between knowing the exact actual execution time beforehand (as clairvoyant algorithms do) and knowing nothing other than the two WCET estimates (as most previously-proposed mixed-criticality scheduling algorithms assume).

Contributions. The main contribution of this paper is to motivate and initiate the exploration of semi-clairvoyant scheduling for mixed-criticality systems. To this end we propose a scheduling-theoretic framework for the consideration of semi-clairvoyance, and derive several results concerning semi-clairvoyant scheduling (the results we have obtained are enumerated in Section II-D, after the appropriate background definitions and concepts are introduced); these results provide an illustration of the capabilities and limitations of semi-clairvoyant scheduling.

Organization. The remainder of this document is organized in the following manner. In Section II we introduce a formal framework for discussing semi-clairvoyance in the context of mixed-criticality scheduling, and provide some additional background to help us better appreciate this context. Sections III–VI provide a thorough exploration of the semi-clairvoyant scheduling of collections of independent mixed-criticality jobs (specific contents of these individual sections are listed in Section II-D). Section VII briefly explores the application of the idea of semi-clairvoyance to systems of recurrent tasks: we show here that the concepts and ideas are directly applicable, and present some preliminary results indicating that there are considerable benefits to semi-clairvoyance in scheduling such systems (as was the case for collections of independent jobs). We conclude in Section VIII with some context, and a few directions for future research.

II. Model

In this section we (i) present the model for mixed-criticality workloads that is widely adopted in the mixed-criticality scheduling literature and that we use in this work (Sec II-A); (ii) propose a classification, based upon the degree of clairvoyance that is available to them, of mixed-criticality scheduling algorithms (Sec II-B); (iii) briefly review some prior results from mixed-criticality scheduling theory (Sec II-C); and (iv) enumerate the contributions contained in this paper (Sec II-D).

A. Our workload model

We will, for the most part, restrict our attention here to *dual*-criticality systems: systems with two distinct criticality levels depicted LO (for *low*) and HI (for *high*) respectively. A problem instance is specified as a collection of n dual-criticality jobs $\mathcal{J} = \{J_1, J_2, ..., J_n\}$ that are to execute upon a single share preemptive processor. Each job J_i is characterized by a tuple of parameters: $J_i = (\chi_i, a_i, [c_i^L, c_i^H], d_i)$, where

- $\chi_i \in \{\text{LO}, \text{HI}\}$ denotes the criticality of the job;
- $a_i \in R^+$ denotes its release time;
- c_i^L and c_i^H denote LO-criticality and HI-criticality estimates of the job's worst-case execution time (WCET) parameter respectively such that $c_i^L \leq c_H^i$; and
- $d_i \in R^+$ denotes its deadline.

System *behavior*. This dual-criticality workload model has the following semantics. Each job J_i of an instance $\mathcal{J} = \{J_1, J_2, ..., J_n\}$ is released at time a_i, has a deadline at d_i, and needs to execute preemptively upon the single shared processor for some duration γ_i. The values $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ are said to define the *behavior* of the instance:

- if $\gamma_i \leq c_i^L$ for all i, then the behavior is said to be a LO-criticality one;
- else if $\gamma_i \leq c_i^H$ for all i, then the behavior is said to be a HI-criticality one;
- else (i.e., $\gamma_i > c_i^H$ for some i) the behavior is said to be erroneous.

Observe that the same instance may exhibit different behaviors during different invocations (runs); indeed, it is this timing non-determinism exhibited by systems executing upon modern processors that motivated the introduction of the Vestal workload model [1].

Correctness criteria. An algorithm for scheduling mixed-criticality instances is said to schedule the instance $\mathcal{J}$ correctly if

1) in every LO-criticality behavior of $\mathcal{J}$, each $J_i \in \mathcal{J}$ completes execution by its deadline; and
2) in every HI-criticality behavior of $\mathcal{J}$, all HI-criticality jobs in $\mathcal{J}$ (i.e., each $J_i \in \mathcal{J}$ with $\chi_i = \text{HI}$) complete execution by their respective deadlines. (Note that LO-criticality jobs are not required to complete execution by their deadlines –or indeed, at all– upon HI-criticality behavior.)

If mixed-criticality scheduling algorithm A schedules instance $\mathcal{J}$ correctly, we say that $\mathcal{J}$ is *A-schedulable*.

B. Mixed-criticality schedulability: A classification

As stated above, the *behavior* of an instance is defined by the durations γ_i for which each job J_i in the instance needs to execute in order to complete; furthermore, different invocations of the instance may result in different behaviors. Mixed-criticality scheduling algorithms (and hence, mixed-criticality schedulability) may be classified according to when the exact values of the γ_i's become revealed to the algorithms.

MC-schedulability. Most mixed-criticality scheduling algorithms that have been studied thus far impose the restriction that the values of the γ_i's are revealed on-line during run-time: the only way for the scheduler learns the value of γ_i is

by actually executing job J_i to completion. Algorithms satisfying this restriction are called MC-scheduling (for "mixed-criticality scheduling") algorithms. Most of the algorithms such as OCBP [10]–[12] and MCEDF [13], [14] that have been developed for scheduling mixed-criticality instances, as well as algorithms such as EDF-VD [15], [16], AMC [17], and MC-Fluid [18], [19] that have been proposed for scheduling systems of mixed-criticality recurrent tasks, fall into this category.

An instance is said to be *MC-schedulable* if it can be scheduled correctly by some MC-scheduling algorithm.

Clairvoyant schedulability. When scheduling an instance $\mathcal{J}$, a clairvoyant scheduler is assumed to know the precise values of the γ_i parameters prior to the arrival of any of the jobs in $\mathcal{J}$. An instance that can be correctly scheduled by some clairvoyant scheduler is said to be *clairvoyant schedulable.* Since the clairvoyant scheduler knows beforehand whether the behavior is a LO-criticality or a HI-criticality (or erroneous) one, observe that clairvoyant schedulability can be easily checked in polynomial time:

1) Determine whether all the jobs can be scheduled using EDF[3] if each executes for its LO-criticality WCET; and
2) Determine whether all the HI-criticality jobs can be scheduled using EDF if each executes for its HI-criticality WCET.

Clairvoyant schedulability is an idealized abstraction that is not realizable for any but the most time-deterministic systems; it is studied primarily as a baseline lower bound against which to compare the performance of schedulers that can actually be implemented in practice (such as the ones that define MC-schedulability).

Semi-clairvoyant schedulability. As mentioned in the introduction, we believe that there is an interesting middle ground between clairvoyant and MC-scheduling algorithms, that we are seeking to represent in the capability required of semi-clairvoyant scheduling algorithms: the ability to know, upon the arrival of job J_i, whether $\gamma_i \leq c_i^L$ will hold or not. Note that semi-clairvoyance does not require that the actual value of γ_i become known when job J_i arrives: all that is required is that the truth or falsehood of the statement "$\gamma_i \leq c_i^L$" become known. We believe that this is a far less onerous requirement than that the exact value of γ_i become known (indeed within the wider context of mixed-criticality scheduling in which different pieces of information are only required to hold at appropriate levels of assurance rather as absolute truths, such a boolean condition can typically be established to a far greater level of assurance than the assurance with which the actual value of γ_i can be determined).

While both MC-schedulability and clairvoyant schedulability have previously been considered in the mixed-criticality scheduling theory literature, to our knowledge the concept of semi-clairvoyant schedulability is new – the definition of this concept, and its analysis in the sections that follow, constitute the main contribution of this paper.

[3]Here as elsewhere in this paper, we are using the well-known result that EDF –the Earliest Deadline First scheduling algorithm– is optimal for executing collections of independent jobs to completion upon a preemptive uniprocessor [20], [21].

C. Some Prior Results

As mentioned above, the idealized abstraction of clairvoyant schedulability for mixed-criticality instances was introduced [9], [10] primarily to serve as a baseline against which to compare the performance of realizable scheduling algorithms. Such comparison has traditionally been quantified via the *speedup factor* metric: an algorithm A has speedup factor f, $f \geq 1$, if, given any instance that is clairvoyantly schedulable upon a particular processor, it is able to schedule the same instance upon a processor that is f times as fast. (Thus smaller values of f are better: an algorithm with speedup 1 is said to be *speedup optimal*.)

The following prior results concerning MC-schedulability are of interest to us:

1) [9, Theorem 1]: Determining whether an instance is MC-schedulable is NP-hard in the strong sense.
2) [9, Proposition 2]: No MC-schedulable algorithm may have a speedup factor smaller than $\left(\sqrt{5}+1\right)/2 \approx 1.618$. (This bound is known to be tight: it was shown in [22] that the algorithm OCBP [10]–[12] has a speedup factor equal to $\left(\sqrt{5}+1\right)/2$.)

D. Summary of Results

We now provide a summary of the findings reported in this paper regarding semi-clairvoyant schedulability:

1) In Section III, we show that no semi-clairvoyant scheduling algorithm may have a speedup factor smaller than $\frac{3}{2}$.
2) In Section IV, we derive a semi-clairvoyant scheduling algorithm that we call LPSC; in Section V, we prove that this algorithm is optimal in the following sense: if an instance can be scheduled correctly by any semi-clairvoyant scheduling algorithm, then it is scheduled correctly by LPSC.
3) In Section VI we show that the speedup bound of LPSC is $\frac{3}{2}$, this result, in conjunction with the lower bound of $\frac{3}{2}$ shown in Section III, establishes that $\frac{3}{2}$ is in fact a tight bound on the speedup of any semi-clairvoyant algorithm.

These results provide stark evidence of the benefits of semi-clairvoyance over MC-scheduling: the speedup bound vis-à-vis clairvoyant schedulability falls from ≈ 1.618 for MC-schedulability to 1.5, and semi-clairvoyant schedulability can be determined in polynomial time while determining MC-schedulability is intractable — NP-hard in the strong sense. In Section VII, we will extend these results to tasks and show that semi-clairvoyant scheduling of recurrent implicit deadline tasks can also be determined in polynomial time and that the speedup factor for implicit deadline semi-clairvoyant tasks is 1 (as opposed to $4/3$ for MC-schedulability speedup). Hence when practical considerations are favorable and permit the use of semi-clairvoyance, semi-clairvoyant algorithms are recommended for use in scheduling mixed-criticality systems.

III. A Lower Bound on Speedup Factor

We now show that no semi-clairvoyant scheduler may have a speedup factor smaller than $3/2$'s. Consider the instance $\mathcal{J}$ comprising the following three jobs:

Job	χ_i	a_i	c_i^L	c_i^H	d_i
J_1	LO	0	1	-	1
J_2	HI	0	1	1	2
J_3	HI	1	0	1	2

It is evident that this instance is clairvoyantly schedulable upon a unit-speed processor:

1) If $\gamma_3 = 0$, then the instance is exhibiting LO-criticality behavior and the scheduler executes J_1 over the interval $[0, 1)$ and J_2 over the interval $[1, 2)$.
2) If $\gamma_3 > 0$, then the instance is exhibiting HI-criticality (or erroneous) behavior. The scheduler drops J_1 and executes J_2 over the interval $[0, 1)$ and J_3 over the interval $[1, 2)$.

Let us now consider the execution of $\mathcal{J}$ by a semi-clairvoyant scheduler upon a speed-f processor, for some $f \geq 1$. We argue by the following reasoning that J_1 must receive one unit of execution over the interval $[0, 1)$: If it receives < 1 unit of execution and then upon J_3's arrival it is revealed that $\gamma_3 = 0$ (i.e., the instance is exhibiting LO-criticality behavior), the scheduler would have violated the correctness criteria by missing the deadline of the LO-criticality job J_1 in a LO-criticality behavior. Hence J_2 receives at most $(f - 1)$ units of execution over $[0, 1)$, and has at least $(1 - (f - 1))$ units of execution remaining. Suppose now that upon J_3's arrival it is revealed that $\gamma_3 = 1$ —i.e., the instance is exhibiting HI-criticality behavior. The remaining $(1 - (f - 1))$ units of J_2's execution requirement, and all one unit of J_3's execution requirement, must be accommodated over the interval $[1, 2)$; since the processor is of speed f, we need that

$$\begin{aligned} & (1 - (f - 1)) + 1 \leq f \\ \Leftrightarrow \quad & 2 - f + 1 \leq f \\ \Leftrightarrow \quad & 3 \leq 2f \\ \Leftrightarrow \quad & f \geq (3/2) \end{aligned}$$

We have thus shown that this particular 3-job instance $\mathcal{J}$, which is clairvoyantly schedulable upon a unit-speed processor, cannot be scheduled correctly by any semi-clairvoyant scheduler upon a processor of speed $< \frac{3}{2}$. Hence, $\frac{3}{2}$ is a lower bound on the speedup factor for any semi-clairvoyant schduling algorithm.

IV. LPSC: A Semi-Clairvoyant Scheduler

We now present, and prove the correctness of, LPSC, a semi-clairvoyant algorithm for scheduling dual-criticality instances.[4] In Section V we will prove that LPSC is an optimal semi-clairvoyant algorithm: if any semi-clairvoyant scheduler can schedule an instance, then LPSC can as well. In Section VI, we will determine LPSC's speedup factor.

[4]The name LPSC stands for Linear-Programming based Semi-Clairvoyant, and alludes to the fact that the algorithm is based on formulating and solving a linear-program (LP) representation of semi-clairvoyant schedulability.

Prior to detailing the algorithm, we start out providing some intuition behind our approach towards determining semi-clairvoyant schedulability.

Intuition. The intuition behind semi-clairvoyant algorithm design is as follows. Until (and unless) a behavior is recognized as being of HI criticality, we have to ensure that each job J_i receives enough execution in order to be able to receive c_i^L units of execution by its deadline. However, each arrival of a HI-criticality job may signal that the behavior is a HI-criticality one — if this happens, the scheduler is not required to meet any LO-criticality job deadlines, which implies that all the prior execution that LO-criticality jobs had received before this instant is "wasted". (We saw in the example of Section III that if the scheduler had known that J_3 would be released with execution time of 1, then it need not have scheduled J_1 at all.) Therefore, until and unless HI-criticality behavior is signalled, we want to maximize the execution accorded to HI-criticality jobs while also ensuring that we remain capable of meeting all LO-criticality jobs' deadlines (in case the behavior remains a LO-criticality one).

In the remainder of this section we will detail the linear programming (LP) based schedulability test that builds upon this intuition. The test can be thought of as comprising the following steps: given a dual-criticality instance $\mathcal{J}$

1) We define a linear program that is feasible (has a non-empty feasible region) if and only if all LO-criticality behaviors of $\mathcal{J}$ are schedulable.
2) We specify an objective function for this linear program that enforces the intuition discussed above: as long as HI-criticality behavior has not been signaled, the execution of HI-criticality jobs is maximized while ensuring that it remains possible to complete all LO-criticality jobs by their deadlines if the behavior remains a LO-criticality one.
3) We describe a run-time algorithm for LO-criticality behaviors, that implements the solution that is obtained by solving the optimization problem formulated in the steps above.
4) Finally, we validate correctness in all HI-criticality behaviors by multiple simulations of this run-time algorithm; in each simulation, we consider the possibility that a different HI-criticality job's arrival has signaled that the behavior is to be a HI-criticality one, and show that we can then subsequently meet all HI-criticality deadlines (by discarding all remaining LO-criticality jobs that have not yet completed execution).

§1. Linear Program for Schedulability Test. We first sort the release times and deadlines of all the jobs in $\mathcal{J}$: we call these *key instants*. Without loss of generality, let us assume that the earliest arrival time of any job is 0; there are $2n-1$ other key instants for the n jobs in $\mathcal{J}$. Let us label these instants from t_0 to t_{2n-1}. We define $(2n - 1)$ variables : l_i is the variable that represents the duration that is reserved by the scheduler for executing LO-criticality jobs over the interval $[0, t_i)$.

We now state the constraints that define feasible schedules. For all t_i and t_j where $j > i$, define S_{ij}^H to be the collection

of all HI-criticality jobs $J_k \in \mathcal{J}$ with arrival time $a_k \geq t_i$ and deadline $d_k \leq t_j$. Similarly, we define S^L_{ij} to be the collection of all LO-criticality jobs J_k with arrival time $a_k \geq t_i$ and deadline $d_k \leq t_j$.

For all values of $0 \leq i \leq 2n-1$ and for $i < j \leq 2n-1$, define the following constraints:

$$l_j - l_i \geq \sum_{J_k \in S^L ij} c^L_k \quad (1)$$

$$(t_j - t_i) - (l_j - l_i) \geq \sum_{J_k \in S^H ij} c^L_k \quad (2)$$

$$l_i \leq l_{i+1} \quad (3)$$

The first constraint encodes schedulability of LO-criticality jobs: it requires that within each interval $[t_i, t_j)$, there is enough time reserved for LO-criticality jobs to satisfy their computational requirements. The second constraint encodes schedulability of HI-criticality jobs in LO-criticality mode — once we have reserved appropriate amount of time for LO-criticality jobs, there is enough time left in the schedule to satisfy the LO-criticality WCETs of all the HI-criticality jobs. The third constraint simply says that l_i's are well-defined and we have no negative execution. There are $O(n^2)$ constraints.

It should be clear that any assignment of values to the l_i variables satisfying these constraints yields a schedule for all LO-criticality behaviors of the instance $\mathcal{J}$. Conversely, if this set of constraints is infeasible, then some LO-criticality behaviors of $\mathcal{J}$ are not schedulable.

§2. An optimization criterion. Now we specify an objective function to ensure that, in any prefix of the schedule, we reserve as little time as possible for LO-criticality jobs — by reserving less time for LO-criticality jobs, we allocate more time to HI-criticality jobs (as discussed above, when introducing the intuition behind the algorithm). Therefore, the objective function is:

$$\textbf{minimize} \quad \sum_{i=1}^{2n-1} l_i \quad (4)$$

Note that this function minimizes for all prefixes simultaneously; this can be seen from the following reasoning. Since l_i is the reservation for LO-criticality jobs over the interval $[0, t_i)$, it includes the reservations over intervals $[0, t_j)$ for all $t_j < t_i$. Therefore, the allocations to LO-criticality jobs over earlier intervals (i.e., the l_i's for smaller values of i) are counted multiple times in the objective functions and hence the LP places more emphasis on minimizing them.

Since all constraints and the optimization function are linear and there are a polynomial number of constraints, this linear program can be solved in time that is polynomial in the size of $\mathcal{J}$, and it has a solution if and only if the instance is feasible in all LO-criticality behaviors. Let us suppose that it does indeed have a solution, and let $\widehat{l_i}$ denote the value assigned to the variable l_i in this solution. Below, we describe how these values are used during run-time to schedule the instance as long as instance behavior is not indicated as being of HI criticality.

§3. Run-time scheduling in LO-criticality behaviors. An obvious run-time implementation of the schedule suggested by the solution to the linear program would have us reserve, for each $i, 1 \leq i \leq 2n-1$, an interval of duration $\left(\widehat{l_i} - \widehat{l_{i-1}}\right)$ at the end of the interval $[t_{i-1}, t_i)$ for LO-criticality jobs, and to execute HI-criticality jobs (ordered amongst themselves according to EDF) in the reminder of the intervals. However, such an implementation may not be work-conserving since there may not be enough execution pending for HI-criticality jobs to use up all of the interval that is allocated to them. So we instead have the scheduler keep track of the amount of actual LO-criticality execution that happens up to each key instant: let program variable L_i denote the amount of actual LO-criticality execution that occurs over $[0, t_i)$. $L_0 \leftarrow 0$. At key instant t_{i-1} we know the value of L_{i-1}, and reserve an interval of duration $\max\left(\widehat{l_i} - L_{i-1}, 0\right)$ at the end of the interval $[t_{i-1}, t_i)$ for LO-criticality jobs; the rest of the interval is available for the execution of HI-criticality jobs. Within their own allocated intervals, the execution order of the LO-criticality and HI-criticality jobs is determined according to EDF. If the HI-criticality jobs do not use up the entirety of their interval, additional LO-criticality execution may be scheduled (again, according to EDF).

§4. Checking Schedulability for HI-criticality behaviors. Now to verify whether all HI-criticality behaviors are also scheduled correctly, we simply simulate the work-conserving schedule described above multiple times. We repeat the following procedure for each key instant t_i defined by the arrival of a HI-criticality job: Run the LO-criticality schedule described above until time t_i, assuming that $\gamma_j = c^L_j$ for all jobs $J_j \in \mathcal{J}$. At this point, we know exactly which jobs are pending and how much work has been done on each pending job. Assume now that the HI-criticality job that arrives at t_i indicates that it will execute for its HI-criticality WCET; hence, it is now known that this is a HI-criticality behavior, and all remaining LO-criticality jobs are dropped. In addition, all HI-criticality jobs that have not yet arrived are also assumed to execute for a duration equal to their respective HI-criticality WCETs. We then simply simulate EDF starting from time t_i for all the HI-criticality jobs (pending and the future arrivals). If all jobs complete by their deadlines in all these simulations, then we declare that the instance $\mathcal{J}$ is schedulable using LPSC. Otherwise, we conclude that $\mathcal{J}$ is not schedulable using LPSC.

Since we must simulate for $O(n)$ values of t_i, we run $O(n)$ simulations. Each simulation takes $O(n^2)$ time if we run it naively, for the total time of $O(n^3)$ to check schedulability in HI-criticality mode. (We can of course speed things up – e.g., the individual simulations could be implemented to have $O(n \log n)$ run-time complexity by using priority queues [23], but the naive $O(n^3)$ algorithm suffices to establish that LPSC has polynomial running time since the linear program can be solved in polynomial time.)

V. THE OPTIMALITY OF LPSC

We will now prove that if a set of jobs $\mathcal{J}$ is schedulable by any semi-clairvoyant scheduler, then LPSC also successfully schedules it.

The following observation should be relatively obvious from the constraints of the linear program since the linear program simply checks that for every interval, there is sufficient time in the interval to satisfy the demand in this interval.

Observation 1. *If the linear program described in Section IV is infeasible, then no scheduler can schedule the set $\mathcal{J}$ of jobs in low-criticality behavior.*

Therefore, in particular, if the linear program is infeasible, then no semi-clairvoyant or clairvoyant scheduler can schedule $\mathcal{J}$ in low-criticality behavior. We must now argue that if LPSC declares that a set of jobs $\mathcal{J}$ is infeasible (while checking schedulability in HI-criticality mode via simulations as described in Section IV), then no other semi-clairvoyant schedule can guarantee that it can schedule $\mathcal{J}$. In the future, we will refer to LPSC when used in mathematical notation as A — in other words, $l_i(A)$ are the values of variables calculated by the linear program described in Section IV; the actual LO-criticality work done by a LPSC by time t_i for $\mathcal{J}$ is $L_i(A)$; and the actual high-criticality work done by LPSC at time t_i is $H_i(A)$.

We will compare LPSC with a hypothetical scheduler B which, when running with LO-criticality behavior, executes $L_i(B)$ work from LO-criticality jobs and $H_i(B)$ work from HI-criticality jobs by time t_i (between time 0 and t_i) for all i. Since all feasible schedulers must satisfy the constraints of the linear program described above, we can state the following observation about $L_i(B)$s.

Observation 2. *For any scheduler B that can schedule a set of jobs $\mathcal{J}$ in low-criticality mode, the LP constraints described in Inequalities 1, 2, and 3 must be satisfied if we substitute l_i's in the constraint with the corresponding $L_i(B)$'s.*

Without loss of generality, we will only consider work-conserving schedulers for B, since for preemptive, single processor scheduling, work-conserving schedulers dominate non-work-conserving schedulers.

Recall that, given a set of jobs $\mathcal{J}$, the linear program generates a solution $l_i(A)$ for all t_i.

Lemma 1. *Say we have a feasible schedule B. For all i, we have $l_i(A) \leq L_i(B)$.*

Proof. From Observation 2, we know that $L_i(B)$'s constitute a feasible solution to the LP constraints described above. Assume for contradiction that there is some feasible solution B with $L_k(B) < l_k(A)$ for some values of k. Now we claim that we can generate a new solution A' which is feasible and we have $\sum_{i=1}^{2n-1} l_i(A') < \sum_{i=1}^{2n-1} l_i(A)$. If we show this, we will reach a contradiction since, by definition, A is the solution of the LP; therefore the optimization function must take the minimum value for A.

We set $l_i(A') = \min\{l_i(A), L_i(B)\}$ for all i. Since there is at least one k for which $L_k(B) < l_k(A)$, it is clear that $\sum_{i=1}^{2n-1} l_i(A') < \sum_{i=1}^{2n-1} l_i(A)$. We now argue that A' is feasible — that is, all constraints are satisfied for $l_i(A')$'s.

We first consider the Inequality 1: we must show that this constraint is satisfied for all values of i and j. We have two cases:

Case 1: $l_i(A') = l_i(A)$ and $l_j(A') = l_j(A)$. The constraint is clearly satisfied since $l_i(A)$ and $l_j(A)$ satisfy the corresponding constraint. The case where $l_i(A') = L_i(B)$ and $l_j(A') = L_j(B)$ is similar since B also satisfies all the constraints of the linear program and therefore $L_i(B)$ and $L_j(B)$ satisfy the corresponding constraint.

Case 2: $l_i(A') = l_i(A) \leq L_i(B)$ and $l_j(A') = L_j(B) \leq l_j(A)$. Lets look at the LHS of the corresponding constraint for A': $l_j(A') - l_i(A') = L_j(B) - l_i(A) \geq L_j(B) - L_i(B)$. Therefore, this constraint is satisfied for A' since the same constraint is satisfied in B. The case where $l_i(A') = L_i(B) \leq l_i(A)$ and $l_j(A') = l_j(A) \leq L_j(A)$ is symmetric.

We can show that the constraint in Inequality 2 is satisfied using a similar case analysis. Again, as in Case 1 above, if both $l_i(A')$ and $l_j(A')$ for some values of i and j take their value from the same solution, either A or B, then the constraint is clearly true since the constraint is true for A and B. For Case 2: say $l_i(A') = l_i(A) \leq L_i(B)$ and $l_j(A') = L_j(B) \leq l_j(A)$. Then we get $l_j(A') - l_i(A') = L_j(B) - l_i(A) \leq l_j(A) - l_i(A)$. Therefore, the left hand side of the constraint is $(t_j - t_i) - (l_j(A') - l_i(A')) \geq (t_j - t_i) - (l_j(A) - l_i(A))$. Therefore, this constraint is satisfied in A' since it is satisfied in A. The case where $l_i(A') = L_i(B) \leq l_i(A)$ and $l_j(A') = l_j(A) \leq L_j(B)$ is symmetric.

The Constraints generated from Inequality 3 are clearly satisfied since both A and B are well-formed. □

Recall that the actual LO-criticality work and HI-criticality work done by a scheduler S by time t_i is $L_i(S)$ and $H_i(S)$ respectively. We now argue that LPSC does the minimum possible low-criticality work for a work conserving scheduler and therefore, maximum possible high-criticality work for any prefix. This is not directly implied by the previous lemma, since recall that, the actual low-criticality work done by the work-conserving scheduler in LPSC, $L_i(A)$ can exceed the values computed by the linear program $l_i(A)$

Lemma 2. *Say A does $L_i(A)$ work by time t_i and B does $L_i(B)$ work by time t_i. $L_i(A) \leq L_i(B)$.*

Proof. If $L_i(A) = l_i(A)$, then the result is a direct consequence of Lemma 1.

Now we prove it by induction on time — in particular, on key instants $t_0, t_1, ...$ The statement is trivially true for L_0. Assume, as the inductive hypothesis, that by time t_{i-1}, we have $L_{i-1}(A) \leq L_{i-1}(B)$. Now let us look at interval t_{i-1} to t_i. There are a few cases:

1) At the end of this interval $L_i(A) = l_i(A)$ — then we are done.

2) A spent the entire interval doing work on HI-criticality jobs. In this case, we have $L_i(A) = L_{i-1}(A) \leq L_{i-1}(B)$ by the inductive hypothesis. Since, $L_{i-1}(B) \leq L_i(B)$, we are done.
3) Recall how the work-conserving scheduler described in Section IV works. It will reserve $l_i(A) - L_{i-1}(A)$ time at the end of the interval for LO-criticality jobs and try to schedule HI-criticality jobs in the remaining interval. Therefore, if $L_i(A) > l_i(A)$ and A did not spend the entire interval on HI-criticality jobs, then at some point in this interval, there were no pending HI-criticality jobs for A. Therefore, at time t_i, no HI-criticality jobs were pending (since all job arrival times are key instants, a new job cannot arrive between consecutive key instants t_{i-1} and t_i). Therefore, since A has completed all available high-criticality work by time t_i, we have $H_i(A) \geq H_i(B)$. Since A and B are both work-conserving, the total work done by each is equal — that is, $H_i(A) + L_i(A) = H_i(B) + L_i(B)$. Therefore, we have $L_i(A) \leq L_i(B)$.

□

The following corollary is obvious from the Lemma 2 since both A and B are work conserving and therefore, we have $H_i(A) + L_i(A) = H_i(B) + L_i(B)$.

Corollary 1. *For all t_i, we have $H_i(A) \geq H_i(B)$.*

We can now prove the main theorem which shows that if any semi-clairvoyant scheduler can schedule a set of jobs, then LPSC can also schedule them.

Theorem 1. *Say LPSC declares a set of jobs $\mathcal{J}$ unschedulable. Then no semi-clairvoyant scheduler can guarantee that it will correctly schedule this set of jobs.*

Proof. Consider the low-criticality mode. Observation 1 says that if the linear program is infeasible, then no scheduler can schedule $\mathcal{J}$ in low-criticality behavior.

Now consider the high-criticality mode. Say that there is a set of jobs such that LPSC finds the solution A to the linear program, but when it simulates the algorithm to check high-criticality schedulability, it declares failure. Therefore, the following must be true. There is some transition time t_j such that all high-criticality jobs released before time t_j had $\gamma_i = c_i^L$ and all high-criticality jobs released at or after time t_j had $\gamma_i = c_i^H$ and when LPSC simulated its scheduler for this scenario, some HI-criticality job missed its deadline. Consider for contradiction that some other semi-clairvoyant scheduler B can schedule this set of jobs.

Say this deadline miss occurred at time t_k. From Corollary 1, we know that $H_j(A) \geq H_j(B)$. In addition, say that $H_j(A, k)$ (and correspondingly $H_j(B, k)$) is the work done by A on jobs with deadlines before time t_k. Since A uses EDF for HI-criticality jobs within their allocated intervals, we can still see that $H_i(A, k) \geq H_j(B, k)$. Therefore, at the transition time A has less pending work to do before time k than B. After transition, A and B both get identical work to do in the worst case, since they must both be able to schedule all subsequent HI-criticality jobs assuming they all have $\gamma_i = c_i^H$. Therefore, if A cannot complete all its pending work by the deadline t_k, then neither can B. □

VI. A Speedup Bound for LPSC

We will now compare our scheduler to the clairvoyant scheduler and prove a speedup bound. In particular, we will show that LPSC has the optimal speedup bound of $3/2$. In Section III, we showed via an example that all semi-clairvoyant schedulers require speedup of at least $3/2$. Therefore, this shows that no semi-clairvoyant scheduler is better than LPSC with respect to speedup.

We first state a structural property about LPSC.

Lemma 3. *A job J_l with deadline $t_l > t_k$ can not interfere with a HI-criticality job J_k with deadline t_k.*

Proof. First consider an HI-criticality job J_l. LPSC schedules all HI-criticality jobs using EDF in their allocated intervals. Therefore, a job with later deadline should not interfere with a job with earlier deadline. Now say J_l is an LO-criticality job. Assume for contradiction that, in A, job J_l interferes with J_k — that is, J_l executes after J_k was released but before J_k completes. Lets say that J_l executes at some time t to $t+1$ and J_k executes at some time t' to $t'+1$ where $t' > t$. We will do an exchange argument. In particular, we can generate another feasible scheduler B where J_k executes from t to $t+1$ and J_l executes from t' to $t'+1$. After this swap, new scheduler has $L_{t+1}(B) < L_{t+1}(A)$. However, this is a contradiction due to Lemma 2.

Note that we are abusing notation here a little since L_t's are only defined for key instants, and $t+1$ may not be a key instant. However, recall that within each interval between key instants — t_i to t_{i+1}, all HI-criticality jobs execute earlier in the interval than all LO-criticality jobs. Therefore, t and t' must be in different intervals to have the sort of interference described above. Say t is in interval t_i to t_{i+1} and t' is in interval t_j to t_{j+1} for $j > i$. In this case, after we swap execution for J_l and J_k, we have $L_{i+1}(B) < L_{i+1}(A)$, which is now really a contradiction from Lemma 2. □

We now prove a useful structural lemma about LPSC. Again, the notation is the same: $l_i(A)$ is the solution computed by the linear program for time t_i while $L_i(A)$ is the actual execution completed by the LPSC work-conserving scheduler based on solution A at time t_i. We give different speeds $s \geq 1$ to LPSC; therefore, we can generalize $L_i(A, s)$ is the amount of work done on LO-criticality jobs by time t_i by the work-conserving scheduler LPSC described in Section IV. We know that $L_i(A, s) \geq l_i(A)$ for all i since $L_i(A) \geq l_i(A)$ and the scheduler should only be able to do more work with higher speed.

Lemma 4. *Consider a key instant t_j and consider any key instant t_i such that: (1) $t_i \leq t_j$; and (2) between time t_i and t_j, there are always some pending HI-criticality job(s). That is, there is some HI-criticality work that has arrived but not completed at every instant between time t_i and t_j. Then, at*

least one (or both) of the following is true:(1) $L_i(A,s) = L_j(A,s)$ *or (2)* $L_j(A,s) = l_j(A)$.

Proof. We prove this by induction on key instants starting from t_i — that is, we start with $t_j = t_i$ and then keep incrementing t_j. If $t_i = t_j$, the statement is trivially true since the first condition holds. For inductive hypothesis, say that the statement is true for t_{j-1} — that is, $L_i(A) = L_{j-1}(A,s)$ or $L_{j-1}(A,s) = l_{j-1}(A)$ or both.

Recall the scheduling algorithm. Within an interval t_{j-1} to t_j, it reserves $\max\{l_j(A) - L_{j-1}(A,s), 0\}$ time for LO-criticality jobs and allocates the remaining time to HI-criticality jobs. It only gives more time LO-criticality jobs if there are no pending HI-criticality jobs. However, between time t_{j-1} and t_j, there are definitely pending HI-criticality jobs by definition of t_i. Therefore, if $\max\{l_j(A) - L_{j-1}(A,s), 0\} > 0$, then the scheduler will allocate exactly $l_j(A) - L_{j-1}(A,s)$ time to LO-criticality jobs and therefore, $L_j(A,s) = l_j(A)$. If $\max\{l_j(A) - L_{j-1}(A,s), 0\} = 0$, then the scheduler will not allocate any time to LO-criticality jobs, and we will have $L_j(A) = L_{j-1}(A,s) = L_i(A,s)$ by inductive hypothesis. □

We now start comparing A with some speed s with an optimal scheduler O with speed 1. Note that even with speed s, LPSC will compute l_i's using the linear program assuming speed 1. The speed comes into play only while LPSC is checking HI-criticality schedulability via simulation. Therefore, Lemma 1 still holds. However, since O has a different speed than A, Lemma 2 and Corollary 1 no longer directly apply since L_i and H_i quantities are based on actual execution and not only on the quantities computed by the linear program.

Therefore, we first define a specific period when a version of the lemma does apply. In order to do so, we first generalize L and H quantities to intervals. In particular $L_{ij}(S,s)$ and $H_{ij}(S,s)$ for any scheduler S is the amount of work of low-criticality and high-criticality jobs respectively done by S between time instants t_i and t_j by S with speed s. If the s parameter is omitted, the speed is 1.

We now define an important quantity for our proof. For any key instant t_j while running LPSC scheduler A with speed s, we define a *pivotal instant* pivot(t_j) as $t_i \leq t_j$ if the following are true:

1) The scheduler A remains busy from time t_i to t_j.
2) No high-criticality job is pending at time t_i.
3) Between time t_i and t_j, we have $L_{ij}(A,s) \leq L_{ij}(O)$ when A is in low-criticality mode and O is scheduling the jobs in the low-criticality behaviour.

If these three properties are true for multiple instants, we can pick any of them as the pivot — all we care about is that we can find some instant $t_i \leq t_j$ when all these properties are true. The following lemma shows that for any key instant t_j, there exists a pivotal instant t_i.

Lemma 5. *For any key instant* t_j*, we can find pivot*$(t_j) \leq t_j$.

Proof. If there are no pending HI-criticality jobs at time t_j, then the pivot$(t_j) = t_j$ since all three properties hold. Otherwise, consider time t_i which is the latest time before t_j when A with speed s had an idle instant — since A is work-conserving, this means that no job was pending at the time. Therefore, properties 1 and 2 from the above definition are automatically satisfied since, by definition, the scheduler has no pending jobs (therefore, no pending HI-criticality jobs) at time t_i and this was the last such instant. We must consider two cases:

Case 1: $L_j(A,s) = l_j(A)$**:** Recall that $l_j(A)$ is the solution computed by the linear program. In this case, from Lemma 1 we know that $L_j(A,s) = l_j(A) \leq L_j(O)$. In addition, since there are no pending jobs for A at time t_i, we know that $L_i(A,s) \geq L_i(O)$. Therefore, $L_{ij}(A,s) = L_j(A,s) - L_i(A,s) \leq L_j(O) - L_i(O) = L_{ij}(O)$. In this case, we have found the pivotal instant pivot$(t_j) = t_i$.

Case 2:$L_j(A,s) > l_j(A)$ Now we find a different pivotal instant — in particular, we consider the latest time t_l before t_j but potentially after t_i when no high-criticality jobs were pending (but low-criticality jobs may be pending). Again, properties 1 and 2 of pivotal instant are satisfied by definition. In this case, from Lemma 4, since we have $L_j(A,s) > l_j(A)$, we know that $L_j(A,s) = L_l(A,s)$. Therefore, we have $L_{lj}(A,s) = 0 \leq L_{ij}(O)$. We can then pick the pivotal instant as pivot$(t_j) = t_l$.

Since we have demonstrated how to find the pivotal instant for all time key instants t_j, a pivotal instant always exists for all key instants. □

We can now prove the speedup bound. We will do so by showing that if O can meet all deadlines under both LO-criticality and HI-criticality behaviors, then A with speed $3/2$ can meet all deadlines also no matter what the behavior of the system.

Theorem 2. *Our algorithm has a speedup factor of* $3/2$ *against a clairvoyant optimal.*

Proof. If the system never enters high-criticality mode A can schedule anything that O can schedule even with speed 1 since they both have to obey the LP constraints.

Assume for contradiction that, there is some job set $\mathcal{J}$ which is clairvoyant schedulable but A with speed $3/2$ transitions to HI-criticality mode at the release time of some HI-criticality job at time t_j and subsequently misses a deadline at time $t_k > t_j$. This deadline miss must be for a high-criticality job since we have discarded all LO-criticality jobs at the time of transition. In addition, A with speed s must remain busy between time t_j and t_k — if it finishes all pending work at some time t_l where $t_j \leq t_l < t_k$, then between t_l and t_k, A has to only complete the jobs that arrive after t_l using EDF and O has to also (at least) do this. Therefore, if O can meet all deadlines, so can A even with speed 1.

We know, by Lemma 3, that no low- or high-criticality job with a later deadline can interfere with a high-criticality job with an earlier deadline. Therefore, we need only consider jobs

with deadlines before t_k. Therefore, we will assume that $\mathcal{J}$ only contains jobs with deadlines before t_k and not consider any jobs with deadline after t_k either for O or for A.

We first define t_i as the pivotal instant for t_j. Therefore, by definition of pivotal instant, we have

$$L_{ij}(A, s) \leq L_{ij}(O) \tag{5}$$

In addition, since O executes $L_{ij}(O)$ work between times t_i and t_j, we have, by definition

$$L_{ij}(O) \leq t_j - t_i \tag{6}$$

Now we generate some constraints on the work done by O with speed 1. Note that O must be able to schedule $\mathcal{J}$ under both low- and high-criticality behaviors of the system. Note that the system transitions at time t_j. Therefore, all the high-criticality jobs that are released before t_j execute for their low-criticality execution time. Say H_{ij}^L is the total low-criticality work of all the high-criticality jobs that were released between time t_i and t_j. (Note that they all have deadlines before t_k since we do not need to consider any jobs with deadlines after.) In the low criticality mode, O has to schedule all this work between time t_i and t_k. Therefore, we have

$$L_{ij}(O) + H_{ij}^L \leq t_k - t_i \tag{7}$$

Now we consider O's schedule in high-criticality behavior — recall that O does not schedule any low-criticality jobs at all in this behavior. Now say H_{jk}^H is the total high-criticality work of all the jobs released between time t_j and t_k (with deadline at most t_k, since we ignore all jobs with later deadlines entirely). In the high-criticality mode O must schedule these jobs within time t_j and t_k even if they all execute for their high-criticality executions. Therefore, we have

$$H_{jk}^H \leq t_k - t_j \tag{8}$$

In addition, in high-criticality mode, O ignores all low criticality jobs, but must still schedule all the high criticality jobs. Therefore, we have

$$H_{jk}^H + H_{ij}^L \leq t_k - t_i \tag{9}$$

Therefore, if we add the Inequalities 6, 7, 8 and 9, we get

$$2(L_{ij}(O) + H_{ij}^L + H_{jk}^H) \leq 3(t_k - t_i) \tag{10}$$

Substituting from Inequality 5

$$L_{ij}(A, s) + H_{ij}^L + H_{jk}^H \leq 3/2(t_k - t_i) \tag{11}$$

Since no HI-criticality work is pending at time t_i by definition of pivotal instant, the LHS of the above equation represents the maximum total amount of work A may have to complete between time t_i and t_k. Therefore, if the system remains busy for this entire interval, then it can complete all this work in time and therefore, it can not miss a deadline. By definition, since t_i is the pivotal instant for t_j, the system remains busy between t_i and t_j. In addition, as we argued before, it also remains busy between t_j and t_k. Hence, we have reached a contradiction and A with speed $3/2$ can not miss a deadline if the set of jobs is clairvoyant schedulable. □

1) Each τ_i initially executes at a rate θ_i^L.
2) If a job of task τ_i has not completed upon receiving C_i^L units of execution (i.e., having executed for (C_i^L/θ_i^L) time), then
   - All LO-criticality tasks are immediately discarded, and
   - Each HI-criticality task henceforth executes at a rate θ_i^H.

Fig. 1. The run-time scheduling strategy used by Algorithm MC-Fluid [18], [19].

VII. Semi-Clairvoyant Scheduling of Dual-Criticality Tasks

In this section we briefly consider the scheduling of systems of independent dual-criticality implicit-deadline sporadic tasks upon a shared preemptive processor. As with jobs, the MC-schedulability (i.e., non-clairvoyant schedulability) of such systems has been extensively studied; in particular, it is known that (i) checking MC-schedulability of implicit-deadline periodic and sporadic task systems is NP-Hard [22]; and (ii) $4/3$ is a lower bound on the speedup factor of any MC-scheduling algorithm [16].

A. Model and Definitions

We assume that a dual-criticality implicit-deadline sporadic task τ_i is characterized by the parameters $(T_i, C_i^L, C_i^H, \chi_i)$, where $\chi_i \in \{\text{LO}, \text{HI}\}$ denotes its criticality, C_i^L and C_i^H its LO and HI criticality WCETs, and T_i its period. We require that $C_i^L \leq C_i^H$. We let τ denote a collection of n tasks that are to be scheduled upon a preemptive unit-speed processor.

The *correctness* of algorithms for scheduling mixed-criticality task systems is defined analogously to the correctness of algorithms for scheduling jobs: a correct algorithm must meet deadlines of all the jobs in all LO-criticality system behaviors, and deadlines of all HI-criitcality jobs in all HI-criticality system behaviors.

B. Fluid scheduling of dual-criticality systems

The MC-Fluid scheduling algorithm [18], [19] is a non-clairvoyant scheduling (i.e., MC-scheduling) algorithm that was designed for scheduling dual-criticality implicit-deadline sporadic task systems upon identical multiprocessor platforms under the *fluid scheduling* model, which allows for schedules in which individual tasks may be assigned a fraction ≤ 1 of a processor (rather than an entire processor, or none) at each instant in time. MC-Fluid operates in the following manner. Prior to run-time, it computes LO-criticality and HI-criticality *execution rates* θ_i^L and θ_i^H for each task $\tau_i \in \tau$ such that the run-time scheduling algorithm depicted in Figure 1 constitutes a correct scheduling strategy for τ. An algorithm for computing suitable values for the θ_i^L and θ_i^H parameters is presented in [18], and a somewhat simpler algorithm subsequently derived in [19], and shown to have a speedup factor of $\frac{4}{3}$.

1) Each job of LO-criticality task τ_i initially executes at a rate θ_i^L.
2) When a job of a HI-criticality task τ_i arrives
   - **If** it indicates that it needs $\leq C_i^L$ units of execution, then it executes at a rate θ_i^L.
   - **Else** (i.e., it indicates a need for $> C_i^L$ units of execution)
     - it executes at a rate θ_i^H
     - all LO-criticality jobs are immediately discarded, and none will be admitted in the future

Fig. 2. Semi-clairvoyant scheduler: run-time scheduling

C. An Optimal Semi-Clairvoyant Algorithm

We now show that the ideas behind MC-Fluid are easily adapted to give us an *optimal* semi-clairvoyant algorithm for dual criticality implicit-deadline tasks upon uniprocessors. Observe first that for an implicit-deadline sporadic task system τ to be schedulable by any scheduler (including a clairvoyant one), it is necessary that the following conditions hold:

$$\sum_{\tau_i \in \tau} \frac{C_i^L}{T_i} \leq 1 \quad (12)$$

$$\sum_{\tau_i \in \tau \bigwedge \chi_i = \text{HI}} \frac{C_i^H}{T_i} \leq 1 \quad (13)$$

The schedulability test associated with our semi-clairvoyant scheduling algorithm is straightforward: any task system τ satisfying the conditions of Inequalities 12 and 13 will be correctly scheduled by our algorithm. The associated run-time strategy is depicted in Figure 2. Our algorithm assigns the θ_i^L and θ_i^H execution rates as follows:

$$\begin{aligned} \theta_i^L &\leftarrow C_i^L/T_i \\ \theta_i^H &\leftarrow C_i^H/T_i \end{aligned}$$

D. Proof of Correctness and Optimality

We now show that this is a correct semi-clairvoyant algorithm: if the schedulability test admits a task system τ (i.e., as long as τ satisfies the conditions in Inequalities 12 and 13), then it will schedule τ correctly.

Lemma 6. *All jobs meet their deadlines in all* LO*-criticality behaviors.*

Proof. This is relatively obvious from the manner in which the θ_i^L execution rates are assigned. Between their arrival time and their deadline, all jobs of task τ_i receive $\theta_i^L \times T_i = C_i^L$ execution, which is sufficient to meet their execution requirement in any LO-criticality behavior. □

Lemma 7. *All* HI*-criticality jobs meet their deadlines in all* HI*-criticality behaviors.*

Proof. All jobs of HI-criticality task τ_i that are released with execution demand larger than C_i^L are assigned an execution rate of θ_i^H. Therefore, they receive an execution of $\theta_i^H \times T_i = C_i^H$ between their arrival time and their deadline, which is sufficient to complete their high-criticality work. □

Lemma 8. *At all instants, the execution rates assigned to all the jobs in the system sum to at most 1.*

Proof. As long as no job has indicated, upon its arrival, that the current behavior is a HI-criticality one, we have the total execution rate of $\sum \theta_i^L \leq 1$. Once some job indicates upon arrival that that the behavior is of HI criticality, the system "transitions" and all jobs of low-criticality tasks are discarded. At this point, some high-criticality jobs may still have execution rates θ_i^L (those with execution demand $\leq C_i^L$). However, we know that $\theta_i^L \leq \theta_i^H$. Therefore, the sum of all execution rates can not be larger than $\sum_{\chi_i = H} \theta_i^H \leq 1$. □

Combining Lemmas 6, 7 and 8 gives us the proof of correctness. Optimality is obvious since the schedulability test admits all task sets that an optimal clairvoyant scheduler can schedule.

VIII. Conclusions

In this paper, we have introduced the notion of semi-clairvoyant scheduling for mixed-criticality workloads, and have contrasted it with both clairvoyant scheduling and on-line or MC-scheduling. While clairvoyant scheduling is an idealized abstraction that is not really implementable, we have argued that some form of semi-clairvoyance may be realizable in many circumstances, (including when a job is implemented in multiple versions with a choice of which specific version to execute upon invocation being made upon job arrival, or when a job with a single implementation has its WCET characterized parametrically and the parameters are only known at the time of the job's release). We have shown that in general semi-clairvoyant scheduling offers superior performance to MC-scheduling: the speedup factor vis-à-vis is smaller (1.5 versus $(\sqrt{5}+1)/2 \approx 1.618$) and schedulability analysis is easier (in polynomial time versus NP-hard in the strong sense). These benefits of semi-clairvoyant scheduling, in conjunction of the possibility of the realization of such algorithms, argues in favor of designing mixed-criticality systems in a manner that enable semi-clairvoyant scheduling, by possibly using parametric WCET-analysis tools and/ or providing multiple implementations of jobs to deal with situations of different criticalities.

We have also briefly explored the applicability of the idea of semi-clairvoyance to systems of recurrent tasks. For implicit deadline dual criticality tasks, we provide an optimal fluid scheduler which can schedule any set of tasks that a clairvoyant algorithm can schedule — again, we see a separation from non-clairvoyant (i.e., MC) scheduling since optimal MC-scheduling is known to be NP-Hard, and $4/3$ has been shown to be a lower bound on the speedup factor of any MC-scheduling algorithm.

There are many open questions in semi-clairvoyant scheduling that merit further study. One concerns generalization to more than two criticality levels: it is not at all clear how the

linear-programming based techniques we used in LPSC can be extended to deal with > 2 criticality levels. Additionally, there are many interesting questions concerning the applicability of semi-clairvoyance upon mixed-criticality workloads comprising recurrent tasks — we have only considered fluid scheduling. Fluid scheduling is not a particularly practical algorithm and it appears to be non-trivial to adapt our algorithm to obtain non-fluid schedules. Finally, this paper addresses uniprocessor platforms and addressing multiprocessor platforms and parallel jobs/tasks remains part of future work.

ACKNOWLEDGEMENTS

This research was supported, in part, by the the National Science Foundation (USA) under Grant Numbers CNS-1618185, CNS-1911460, CCF-1337218 and CCF-1439062, and in part by EPSRC grant MCCps (EP/P003664/1).

REFERENCES

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the Real-Time Systems Symposium*. Tucson, AZ: IEEE Computer Society Press, December 2007, pp. 239–243.

[2] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "How realistic is the mixed-criticality real-time system model?" in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ser. RTNS '15. New York, NY, USA: ACM, 2015, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/2834848.2834869

[3] R. Ernst and M. D. Natale, "Mixed criticality systems - A history of misconceptions?" *IEEE Design & Test*, vol. 33, no. 5, pp. 65–74, 2016. [Online]. Available: http://dx.doi.org/10.1109/MDAT.2016.2594790

[4] S. Baruah, "Mixed-criticality scheduling theory: Scope, promise, and limitations," *IEEE Design Test*, vol. 35, no. 2, pp. 31–37, April 2018.

[5] A. Burns, R. Davis, S. Baruah, and I. Bate, "Robust mixed-criticality systems," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1478–1491, 10 2018.

[6] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 82:1–82:37, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3131347

[7] G. Bernat and A. Burns, "An approach to symbolic worst-case execution time analysis," *IFAC Proceedings Volumes*, vol. 33, no. 7, pp. 43 – 48, 2000, 25th IFAC Workshop on Real-Time Programming (WRTP'2000), Palma, Spain, 17-19 May 2000.

[8] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, 2003, pp. 99–102.

[9] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1140–1152, 2012.

[10] 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)*. IEEE, April 2010.

[11] ——, "Mixed-criticality scheduling: improved resource-augmentation results," in *Proceedings of the ICSA International Conference on Computers and their Applications (CATA)*. IEEE, April 2010.

[12] H. Li, "Scheduling mixed-criticality real-time systems," Ph.D. dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill, 2013.

[13] 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*, ser. ECRTS '13. Paris (France): IEEE Computer Society Press, 2013.

[14] D. Socci, "Scheduling of certifiable mixed-criticality systems," Ph.D. dissertation, University Joseph Fourier (Grenoble), 2016.

[15] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ser. ECRTS '12. Pisa (Italy): IEEE Computer Society, 2012.

[16] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems," *Journal of the ACM*, vol. 62, no. 2, pp. 14:1–14:33, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2699435

[17] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. Vienna, Austria: IEEE Computer Society Press, 2011.

[18] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "MC-Fluid: Fluid model-based mixed-criticality scheduling on multiprocessors," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*, Dec 2014, pp. 41–52.

[19] S. Baruah, A. Easwaran, and Z. Guo, "MC-Fluid: simplified and optimally quantified," in *Real-Time Systems Symposium (RTSS), 2015 IEEE*, Dec 2015.

[20] M. Dertouzos, "Control robotics : the procedural control of physical processors," in *Proceedings of the IFIP Congress*, 1974, pp. 807–813.

[21] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[22] K. Agrawal and S. Baruah, "Intractability issues in mixed-criticality scheduling," in *2018 30th Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, July 2018.

[23] 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*, Washington D.C., May 1988, pp. 42–46.