Achieving temporal isolation in multiprocessor mixed-criticality systems

Sanjoy Baruah
The University of North Carolina
baruah@cs.unc.edu

Alan Burns
The University of York
alan.burns@york.ac.uk

***Abstract*—Upon mixed-criticality environments, the execution of high-criticality functionalities must be protected from interference from the execution of less critical functionalities. A means of achieving this objective upon multiprocessor environments is by forbidding less critical functionalities from executing anywhere upon the platform while more critical functionalities are executing upon any processor. This approach towards ensuring temporal isolation between the different criticalities that are co-implemented upon a common platform is explored in this paper, under both the global and partitioned paradigms of multiprocessor scheduling.**

I. Introduction

Thus far, mixed-criticality scheduling (MCS) theory has primarily concerned itself with the sharing of CPU computing capacity in order to satisfy the computational demand, as characterized by the worst-case execution times (WCET), of pieces of code. However, there are typically many additional resources that are also accessed in a shared manner upon a computing platform, and it is imperative that these resources also be considered in order that the results of MCS research be applicable to the development of actual systems. An interesting approach towards such consideration was advocated by Giannopoulou et al. [4] in the context of multiprocessor platforms: during any given instant in time, all the processors are only allowed to execute code of the same criticality level. This approach has the advantage of ensuring that access to all shared resources (memory buses, cache, etc.) during any time-instant are only from code of the same criticality level; since code of lower criticality are not allowed to execute simultaneously with code of higher criticality, the possibility of less critical code interfering with the execution of more critical code in accessing shared resources is ruled out.

In this paper, we seek to better understand the multiprocessor scheduling of mixed-criticality systems under this constraint of only executing jobs of a particular criticality at any given instant in time. To our knowledge, not much is known such scheduling problems; accordingly, we start out with a very simple model in which there are just two criticality levels (designated, as is standard, as HI and LO), and the workload is represented as a specified collection of independent jobs that all have a common release date and a common deadline. We consider the scheduling of such mixed-criticality systems both when jobs are permitted to migrate between processors (global scheduling), and when inter-processor migration is forbidden (partitioned scheduling). Global scheduling necessarily assumes a preemptive model of execution; for partitioned scheduling, preemption is not needed when all jobs have the same release date and deadline.

Organization. The remainder of this paper is organized as follows. In Section II, we elaborate upon the workload model that will be assumed in the reminder of this paper. We discuss the global scheduling of mixed-criticality task systems represented using this model in Section III, and partitioned scheduling in Section IV.

II. Model

A mixed-criticality *job* $j_i = (\chi_i, a_i, C_i(\text{LO}), C_i(\text{HI}), d_i)$ is characterized by a criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$, a release time a_i, a deadline d_i, and two WCET estimates $c_i(\text{LO})$ and $c_i(\text{HI})$. We assume that $a_i, d_i, C_i(\text{LO})$, and $C_i(\text{HI})$ are all $\in \mathcal{R}^+$, the non-negative real numbers.

We seek to schedule a mixed-criticality instance I, consisting of a given collection of mixed-criticality jobs that all have the same release time (without loss of generality, assumed to be equal to zero) and the same deadline (denoted D) upon a platform of m unit-speed processors. Let I_{LO} denote the LO-criticality jobs in I, and I_{HI} the HI-criticality jobs in I. As is standard in mixed-criticality scheduling, we assume that the exact amount of execution required by a job is not known beforehand. If each job j_i completes upon executing for no more than $C_i(\text{LO})$ units, we say that the system exhibits LO-criticality behavior; if some j_i executes more than $C_i(\text{LO})$, but no more than $C_i(\text{HI})$, units, we say that the system exhibits HI-criticality behavior. The correctness requirement is that all jobs should complete by their deadlines in all LO-criticality behaviors, while only the HI-criticality jobs need to complete by their deadlines in any HI-criticality behavior.

Approach. The over-all approach that we advocate here is to first schedule the HI-criticality jobs during run-time — this can be thought of as a generalization of the *criticality monotonic (CM)* priority-assignment approach, which was previously shown [1] to be optimal for scheduling instances in which all jobs have equal deadlines (such as the instances considered here) upon uniprocessor platforms. If each HI-criticality job signals completion upon having executed for no more than its LO-criticality WCET, we recognize that we are in a LO-criticality behavior and begin execution of the LO-criticality jobs.

Notice that under the advocated approach, LO-criticality jobs only begin to execute after no HI-criticality jobs remain

	χ_i	a_i	$C_i(\text{LO})$	$C_i(\text{HI})$	d_i
j_1	LO	0	6	6	10
j_2	LO	0	6	6	10
j_3	LO	0	6	6	10
j_4	HI	0	2	10	10
j_5	HI	0	2	10	10
j_6	HI	0	4	4	10
j_7	HI	0	4	4	10

TABLE I
AN EXAMPLE MIXED-CRITICALITY TASK INSTANCE.

that need to be executed. The problem of scheduling these LO-criticality jobs therefore becomes a "regular" (i.e., not mixed-criticality) scheduling problem. Hence we can first determine, using techniques from conventional scheduling theory, the minimum duration (the *makespan*) of a schedule for the LO-criticality jobs. Once this makespan Δ is determined, the difference between D and this makespan (i.e., $(D - \Delta)$) is the duration for which the HI-criticality jobs are allowed to execute to completion in any LO-criticality schedule. Determining schedulabilty for the mixed-criticality instance is thus reduced to determining whether I_{HI} can be scheduled in such a manner that

- If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{LO})$, then the schedule makespan is $\leq (D - \Delta)$; and
- If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{HI})$, then the schedule makespan is $\leq D$.

Note that we do not in general know, prior to actually executing the jobs, whether each job will complete within its LO-criticality WCET or not. Hence it is not sufficient to construct two entirely different schedules that separately satisfy these two requirements above; instead, the two schedules must be identical until at least that point in time at which some job executes for more than its LO-criticality WCET. (This observation is further illustrated in the context of global scheduling in Example 1 below.)

III. GLOBAL SCHEDULING

We start out considering the global scheduling of instances of the kind described in Section II above. Given a collection of n jobs with execution requirements $c_1, c_2, \ldots, c_n$, McNaughton [8, page 6] showed as far back as 1959 that the minimum makespan of a preemptive schedule for these jobs on m unit-speed processors is

$$\max\left(\frac{\sum_{i=1}^{n} c_i}{m}, \max_{i=1}^{n}\{c_i\}\right) \qquad (1)$$

A direct application of McNaughton's result yields the conclusion that the minimum makespan Δ for a global preemptive schedule for the jobs in I_{LO} is given by

$$\Delta \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{LO}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{LO}}\{C_i(\text{LO})\}\right) \qquad (2)$$

Hence in any LO-criticality behavior it is necessary that the HI-criticality jobs in I — i.e., the jobs in I_{HI} — must be scheduled to have a makespan no greater than $(D - \Delta)$:

$$\max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{HI}}\{C_i(\text{LO})\}\right) \leq D - \Delta\ . \qquad (3)$$

(Here, the makespan bound on the LHS follows again from a direct application of McNaughton's result.) Additionally, in order to ensure correctness in any HI-criticality behavior it is necessary that the makespan of I_{HI} when each job executes for its HI-criticality WCET not exceed D:

$$\max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{HI})}{m}, \max_{\chi_i=\text{HI}}\{C_i(\text{HI})\}\right) \leq D \qquad (4)$$

(where the LHS is again obtained using McNoughton's result.)

One might be tempted to conclude that the conjunction of Conditions 3 and 4 yields a sufficient schedulability test. However, this conclusion is erroneous, as is illustrated in Example 1 below.

Example 1: Consider the scheduling of the mixed-criticality instance of Table I upon 3 unit-speed processors. For this instance, it may be validated that

- By Equation 2, Δ evaluates to $\max(\frac{6+6+6}{3}, 6)$ or 6.
- The LHS of Condition 3 evaluates to $\max(\frac{2+2+4+4}{3}, 4)$, or 4. Condition 3 therefore evaluates to true.
- The LHS of Condition 4 evaluates to $\max(\frac{10+10+4+4}{3}, 10)$, or 10. Condition 4 therefore also evaluates to true.

However for this example, the schedule that causes Condition 3 to evaluate to true *must* have jobs j_6 and j_7 execute throughout the interval $[0, 4)$, while the one that causes Condition 4 to evaluate to true must have j_4 and j_5 execute throughout the interval $[0, 10)$ — see Figure 1. Since we only have three processors available, during any given execution of the instance at least one of the four jobs j_4–j_7 could not have been executing throughout the interval $[0, 2)$.

- If one of $\{j_4, j_5\}$ did not execute throughout $[0, 2)$ and the behavior of the system turns out to be a HI-criticality one, then the job not executing throughout $[0, 2)$ will miss its deadline.
- If one of $\{j_6, j_7\}$ did not execute throughout $[0, 2)$ and the behavior of the system turns out to be a LO-criticality one, then the job $\in \{j_6, j_7\}$ not executing throughout $[0, 2)$ will not complete by time-instant 4, thereby delaying the start of the execution of the LO-criticality jobs to beyond time-instant 4. These jobs will then not be able to complete by their common deadline of 10.

■

The example above illustrates that the conjunction of Conditions 3 and 4, with the value of Δ defined according to Equation 2, is a necessary but *not* a sufficient global schedulability test. Below, we will derive a sufficient global schedulability test with run-time that is polynomial in the representation of the input instance; we will then illustrate, in Example 2, how this test does not claim schedulability of the instance from Example 1. This schedulability test is based upon a network flow argument, as follows. We will describe a

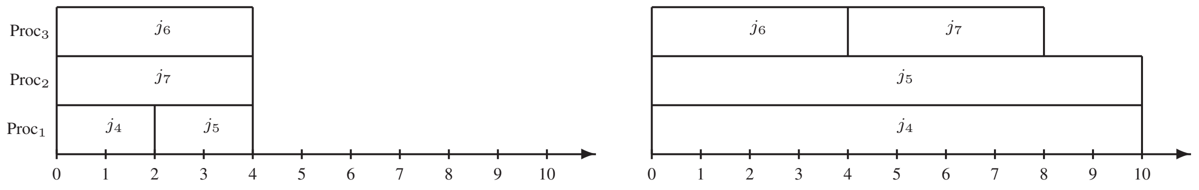


Fig. 1. Schedules for I_{HI} for the task system of Example 1. The left schedule is for a LO-criticality behavior, and has a makespan of four; it thus bears witness to the fact that this mixed-criticality instance satisfies Condition 3. The right schedule is for a HI-criticality behavior – it has a makespan of ten, thereby bearing witness that the instance satisfies Conditions4 as well. Observe that the schedules are *different* at the start: job j_5 does not execute over $[0,2)$ in the left schedule but it does in the right schedule, while job j_7 does not execute over $[0,4)$ in the right schedule but it does in the left schedule.

polynomial-time reduction from any dual-criticality instance I to a weighted digraph G with a designated source vertex and a designated sink vertex, such that flows of a certain size or greater from the source to the sink in G correspond exactly (in a manner that will be made precise) to a valid global schedule for the instance I. Thus, the problem of determining global schedulability is reduced to determining the size of a maximum flow in a graph, which is known to be solvable in polynomial time using, for instance, the Floyd-Fulkerson algorithm [3].

We now describe below the construction of a weighted digraph G from an input instance I. First, we compute the value of Δ for this input instance according to Equation 2. The graph we build is a "layered" one: the vertex set V of G is the union of 6 disjoint sets of vertices $V_0,\ldots,V_5$, and the edge set E of G is the union of 5 disjoint sets of edges $E_0,\ldots,E_4$, where E_i is a subset of $(V_i \times V_{i+1} \times \mathcal{R}^+)$, $0 \le i \le 4$. The digraph G is thus a 6-layered graph — see Figure 2 — in which all edges connect vertices in adjacent layers. The sets of vertices in G are as follows:

$$\begin{aligned}
V_0 &= \{\mathsf{source}\},\\
V_1 &= \{\langle 1, j_i\rangle \mid j_i \in I_{\text{HI}})\},\\
V_2 &= \{\langle 2, j_i, \text{LO}\rangle, \langle 2, j_i, \text{HI}\rangle \mid j_i \in I_{\text{HI}}\},\\
V_3 &= \{\langle 3, j_i, \alpha\rangle, \langle 3, j_i, \beta\rangle \mid j_i \in I_{\text{HI}}\},\\
V_4 &= \{\langle 4, \alpha\rangle, \langle 4, \beta\rangle\}, \text{ and}\\
V_5 &= \{\mathsf{sink}\}.
\end{aligned}$$

Intuitively speaking, each vertex in V_1 represents a HI-criticality job; for each such job, there are two vertices in V_2 representing respectively its LO-criticality execution and the excess execution (beyond its LO-criticaliy WCET) in case of HI-criticality behavior. The vertex $\langle 3, j_i, \alpha\rangle$ will correspond to the amount of execution job j_i actually receives over the interval $[0, D-\Delta)$ – i.e., during the interval within which it must complete execution within any LO-criticality behavior; the vertex $\langle 3, j_i, \beta\rangle$ will correspond to the amount of execution job j_i receives over the interval $[D-\Delta, D)$. The vertices $\langle 4,\alpha\rangle$ and $\langle 4,\beta\rangle$ represent the total amount of execution performed upon the platform during the intervals $[0, D-\Delta)$ and $[D-\Delta, D)$ respectively.

Next, we list the edges in G. An edge is represented by a 3-tuple: for $u, v \in V$ and $w \in \mathcal{R}^+$, the 3-tuple $(u, v, w) \in E$ represents an edge from u to v that has a capacity w. The sets of edges in G are as follows:

$$\begin{aligned}
E_0 &= \{(\mathsf{source}, \langle 1, j_i\rangle, C_i(\text{HI})) \mid j_i \in I_{\text{HI}}\},\\
E_1 &= \{(\langle 1, j_i\rangle, \langle 2, j_i, \text{LO}\rangle, C_i(\text{LO})),\\
&\qquad (\langle 1, j_i\rangle, \langle 2, j_i, \text{HI}\rangle, C_i(\text{HI}) - C_i(\text{LO})) \mid j_i \in I_{\text{HI}}\},\\
E_2 &= \{(\langle 2, j_i, \text{LO}\rangle, \langle 3, j_i, \alpha\rangle, C_i(\text{LO})),\\
&\qquad (\langle 2, j_i, \text{HI}\rangle, \langle 3, j_i, \alpha\rangle, C_i(\text{HI}) - C_i(\text{LO})),\\
&\qquad (\langle 2, j_i, \text{HI}\rangle, \langle 3, j_i, \beta\rangle, C_i(\text{HI}) - C_i(\text{LO})), \mid j_i \in I_{\text{HI}}\},\\
E_3 &= \{(\langle 3, j_i, \alpha\rangle, \langle 4, \alpha\rangle, D-\Delta),\\
&\qquad (\langle 3, j_i, \beta\rangle, \langle 4, \beta\rangle, \Delta) \mid j_i \in I_{\text{HI}}, \text{ and}\\
E_4 &= \{(\langle 4, \alpha\rangle, \mathsf{sink}, m(D-\Delta)), (\langle 4, \Delta\rangle, \mathsf{sink}, m\Delta)\}.
\end{aligned}$$

We now try and explain the intuition behind the construction of G. The maximum flow that we will seek to push from the source to the sink is equal to $\sum_{j_i \in I_{\text{HI}}} C_i(\text{HI})$. Achieving this flow will require that $C_i(\text{HI})$ units of flow go through $\langle 1, j_i\rangle$, which in turn requires that $C_i(\text{LO})$ units of flow go through $\langle 2, j_i, \text{LO}\rangle$, and $(C_i(\text{HI}) - C_i(\text{LO}))$ units of flow go through $\langle 2, j_i, \text{HI}\rangle$, for each $j_i \in I_{\text{HI}}$. This will require that all $C_i(\text{LO})$ units of flow from $\langle 2, j_i, \text{LO}\rangle$ go through $\langle 3, j_i, \alpha\rangle$; the flows from $\langle 2, j_i, \text{HI}\rangle$ through the vertices $\langle 3, j_i, \alpha\rangle$ and $\langle 3, j_i, \beta\rangle$ must sum to $(C_i(\text{HI}) - C_i(\text{LO}))$ units. The global schedule for I_{HI} is determined as follows: ***the amount of execution received by*** j_i ***during*** $[0, D-\Delta)$ ***is equal to the amount of flow through*** $\langle 3, j_i, \alpha\rangle$***; the amount of execution received by*** j_i ***during*** $[D-\Delta, D)$ ***is equal to the amount of flow through*** $\langle 3, j_i, \beta\rangle$**.** Since the outgoing edge from $\langle 3, j_i, \alpha\rangle$ has capacity $(D-\Delta)$, it is assured that j_i is not assigned more execution than can be accommodated upon a single processor; since the outgoing edge from $\langle 4, \alpha\rangle$ is of capacity $m(D-\Delta)$, it is assured that the total execution allocated during $[0, D-\Delta)$ does not exceed the capacity of the m-processor platform to accommodate it. Similarly for the interval $[d-\Delta, D)$: since the outgoing edge from $\langle 3, j_i, \beta\rangle$ has capacity Δ, it is assured that j_i is not assigned more execution than can be accommodated upon a single processor; since the outgoing edge from $\langle 4, \beta \rangle$ is of capacity $m\Delta$, it is assured that the total execution allocated during $[D - \Delta, D)$ does not exceed the capacity of the m-processor platform to accommodate it. Now for both the intervals $[0, D - \Delta)$ and $[D - \Delta, D)$, since no individual job is assigned more execution than the interval duration and the total execution assigned is no more than m times the interval duration, McNaughton's result (Condition 1) can be used to conclude that these executions can be accommodated within the respective intervals.

This above informal argument can be formalized to establish the following lemma; we omit the details.

Theorem 1: If there is a flow of size

$$\sum_{j_i \in I_{\text{HI}}} C_i(\text{HI})$$

in G then there exists a global schedule for the instance I. ■

Example 2: Let us revisit the task system described in Example 1 — for this example instance, we had seen by instantiation of Equation 2 that $\Delta = 6$. The digraph constructed for this task system would require each of j_4–j_7 to transmit at least their corresponding $C_i(\text{LO})$'s, i.e., $2, 2, 4$, and 4, respectively, units of flow through the vertex $\langle 4, \alpha \rangle$, which is do-able since the platform capacity over this interval is $3 \times 4 = 12$. But such a flow completely consumes the platform capacity during $[0, 4)$, which requires that *all* of j_4 and j_5's $(C_i(\text{HI}) - C_i(\text{LO}))$ flows, of $(10 - 2) = 8$ units each, flow through the edges $(\langle 3, j_4, \beta \rangle, \langle 4, \beta \rangle, 6)$ and $(\langle 3, j_5, \beta \rangle, \langle 4, \beta \rangle, 6)$. But such a flow would exceed the capacity of the edge (which is six units), and is therefore not feasible. The digraph constructed for the example instance of Example 1 thus does not permit a flow of size $\sum_{j_i \in I_{\text{HI}}} C_i(\text{HI})$, and Theorem 1, does not declare the instance to be globally schedulable.
■

As previously stated, determining the maximum flow through a graph is a well-studied problem. The Floyd-Fulkerson algorithm [3], first published in 1956, provides an efficient polynomial-time algorithm for solving it. In fact, the Floyd-Fulkerson algorithm is constructive in the sense that it actually constructs the flow – it tells us how much flow goes through each edge in the graph. We can therefore use a flow of the required size, if it exists, to determine how much of each job must be scheduled prior to $(D - \Delta)$ in the LO-criticality schedule, and use this pre-computed schedule as a look-up table to drive the run-time scheduler.

IV. Partitioned scheduling

We now turn to partitioned scheduling, which was the context within which Giannopoulou et al. [4] had initially proposed the paradigm of only executing jobs of one criticality at any instant in time. In partitioned scheduling, we will partition the entire collection of jobs – both the HI-criticality and the LO-criticality ones – amongst the processors prior to run-time. We will also determine some time-instant S, $0 \leq S \leq D$, such that only HI-criticality jobs are executed upon all the processors during $[0, S)$, and only LO-criticality jobs are executed during $[S, D)$. A run-time protocol needs to be defined and supported that will manage the change from HI-criticality to LO-criticality execution. As HI-criticality and LO-criticality jobs cannot execute concurrently, any processor that is still executing a HI-criticality job at some time t must prevent *all* other processors from switching to LO-criticality jobs. Such a protocol would need to be ether affirmative ("its OK to change") or negative ("do not change"):

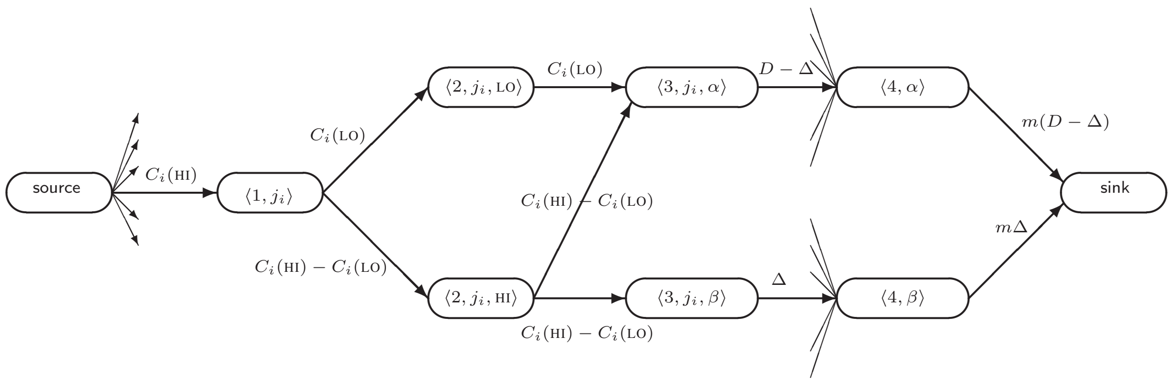
- *affirmative*: each processor broadcasts a message (or writes to shared memory) to say it has completed all its HI-criticality jobs; when each processor has completed its own HI-criticality work and has received $(m - 1)$ such messages it switches to LO-criticality work.
- *negative*: if any processor is still executing its HI-criticality work at time S it broadcasts a message to inform all other processors; any processor that is not in receipt of such a message at time $S{+}\delta$ will move to its LO-criticality work (where δ is a deadline for receipt of the 'no-change' message, determined based upon system parameters such as maximum propagation delay).

In terms of message-count efficiency, the negative message is more effective since normal behavior would result in no messages being sent; whereas the affirmative protocol would generate m broadcast messages. The affirmative protocol is, however, more resilient and less dependent on the temporal behavior of the communication media.

If shared memory is used then a set of flags could indicate the status of each processor. However, spinning on the value of such flags could cause bus contention issues for those processors attempting to complete their HI-criticality work.

We now turn to the problem of partitioning the jobs amongst the processors prior to run-time. Let us first consider the scheduling of just the LO-criticality jobs — i.e., the jobs in I_{LO}. Determining a schedule of minimum makespan for these jobs is equivalent to the bin-packing [6] problem, and is hence highly intractable: NP-hard in the strong sense. Hochbaum and Shmoys [5] have designed a *polynomial-time approximation scheme* (PTAS) for the partitioned scheduling of a collection of jobs to minimize the makespan that behaves as follows. Given any positive constant ϕ, if an optimal algorithm can partition a given task system τ upon m processors each of speed s, then the algorithm in [5] will, in time polynomial in the representation of τ, partition τ upon m processors each of speed $(1 + \phi)s$. This can be thought of as a *resource augmentation* result [7]: the algorithm of [5] can partition, in polynomial time, any task system that can be partitioned upon a given platform by an optimal algorithm, provided it (the algorithm of [5]) is given augmented resources (in terms of faster processors) as compared to the resources available to the optimal algorithm.

We can use the PTAS of [5] to determine in polynomial time, to any desired degree of accuracy, the minimum makespan of any partitioned schedule of the LO-criticality jobs in the instance I. Let Δ_P denote this makespan. Hence



source
$C_i(\text{HI})$
$\langle 1, j_i \rangle$
$C_i(\text{LO})$
$\langle 2, j_i, \text{LO} \rangle$
$C_i(\text{HI}) - C_i(\text{LO})$
$\langle 2, j_i, \text{HI} \rangle$
$C_i(\text{LO})$
$C_i(\text{HI}) - C_i(\text{LO})$
$C_i(\text{HI}) - C_i(\text{LO})$
$\langle 3, j_i, \alpha \rangle$
$\langle 3, j_i, \beta \rangle$
$D - \Delta$
Δ
$\langle 4, \alpha \rangle$
$\langle 4, \beta \rangle$
$m(D - \Delta)$
$m\Delta$
sink

Fig. 2. Digraph construction illustrated. All vertices and edges pertinent to the job j_i are depicted. Additional edges emanate from vertex sink to a vertex $\langle 1, j_\ell \rangle$, for each $j_\ell \in I_{\text{HI}}$; additional edges enter the vertices $\langle 4, \alpha \rangle$ and and $\langle 4, \beta \rangle$ from vertices $\langle 3, j_\ell, \alpha \rangle$ and $\langle 3, j_\ell, \beta \rangle$ respectively, for each $j_\ell \in I_{\text{HI}}$.

to ensure a correct schedule we need to complete scheduling all the HI-criticality jobs in I within the time interval $[0, D - \Delta_P)$; i.e., with a makespan $(D - \Delta_P)$. (The time-instant S mentioned above in the context of the run-time management of the system is therefore equal to $D - \Delta_P$.) Now, determining whether I can be successfully scheduled using partitioned scheduling reduces to determining whether there is a partitioning of just the HI-criticality jobs — i.e., the jobs in I_{HI} — satisfying the properties that

P1. If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{LO})$, then the schedule makespan is $\leq (D - \Delta_P)$; and

P2. If each job $j_i \in I_{\text{HI}}$ executes for no more than $C_i(\text{HI})$, then the schedule makespan is $\leq D$.

(Note that both these properties must be satisfied by a *single* partitioning of the jobs in I_{HI} – it is not sufficient to identify one partitioning that satisfies P1 and another that satisfies P2.)

This partitioning problem turns out to be closely related to the *vector scheduling problem*. Vector scheduling is the natural multi-dimensional generalization of the partitioning problem to minimize makespan, to situations where jobs may use multiple different kinds of resources and the load of a job cannot be described by a single aggregate measure. For example, if jobs have both CPU and memory requirements, their processing requirements are appropriately modeled as two dimensional vectors, where the value along each dimension corresponds to one of the requirements. Clearly, an assignment of vectors to the processors is valid if and only if no processor is overloaded along any dimension (i.e., for any resource). Chekuri and Khanna [2] give a PTAS for solving the vector scheduling problem when the number of dimensions is a constant.

It is not difficult to map our problem of partitioning the HI-criticality jobs (discussed above) to the vector scheduling problem. Each HI-criticality job $j_i \in I_{\text{HI}}$ is modeled as a two-dimensional load vector $\langle C_i(\text{LO}), C_i(\text{HI}) \rangle$, and the capacity constraint for each processor is represented by the vector $\langle (D - \Delta_P), D \rangle$. We can therefore use the PTAS of [2] to determine whether I_{HI} can be partitioned in a manner that satisfies the properties P1 and P2 above, to any desired degree of accuracy in time polynomial in the representation of the instance.

V. Context and Conclusions

Mixed-criticality scheduling (MCS) theory must extend consideration beyond simply CPU computational demand, as characterized by the worst-case execution times (WCET), if it is to be applicable to the development of actual mixed-criticality systems. One interesting approach towards achieving this goal was advocated by Giannopoulou et al. [4] — enforce temporal isolation amongst different criticality levels by only permitting functionalities of a single criticality level to execute at any instant in time. Such inter-criticality temporal isolation ensures that access to all shared resources are only from equally critical functionalities, and thereby rules out the possibility of less critical functionalities compromising the execution of more critical functionalities while accessing shared resources.

We have considered here the design of scheduling algorithms that implement this approach. For a very simple workload model — a dual-criticality system that is represented as a collection of independent jobs that share a common release time and deadline — we have designed asymptotically optimal algorithms for both global and partitioned scheduling:

- For global scheduling, we have designed a polynomial-time sufficient schedulability test that determines whether a given mixed-criticality system is schedulable, and an algorithm that actually constructs a schedule if it is.
- For partitioned scheduling, we have shown that the problem is NP-hard in the strong sense, thereby ruling out the possibility of obtaining optimal polynomial-time algorithms (unless P = NP). We have however obtained what is, from a theoretical perspective, the next best thing – a polynomial-time approximation scheme (PTAS) that determines, in polynomial time, a partitioning of the task system that is as close to being an optimal partitioning algorithm as desired.

The work reported here should be considered to be merely a starting point for research into the particular approach towards mixed-criticality scheduling advocated in [4]. While the PTAS for partitioned scheduling is likely to be the best we can hope for (in asymptotic terms), we do not have a estimate as to how far removed from optimality our global schedulability test is. We also plan to extend the workload model to enable consideration of jobs with different release dates and deadlines, and later to the consideration of recurrent task systems. An orthogonal line of follow-up research concerns the implementation of the global and partitioned approaches presented here – experimentation is needed to determine how they behave upon actual systems.

Acknowledgements

This research is partially supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1409175; ARO grant W911NF-09-1-0535; and ESPRC grant MCC (EP/K011626/1).

References

[1] 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.

[2] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999, pp. 185–194.

[3] L. Ford and D. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, 1956.

[4] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *International Conference on Embedded Software (EMSOFT)*, Montreal, Oct 2013, pp. 17:1–17:15.

[5] D. Hochbaum and D. Shmoys, "Using dual approximation algorithms for scheduling problems: Theoretical and practical results," *Journal of the ACM*, vol. 34, no. 1, pp. 144–162, Jan. 1987.

[6] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[7] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM*, vol. 37, no. 4, pp. 617–643, 2000.

[8] R. McNaughton, "Scheduling with deadlines and loss functions," *Management Science*, vol. 6, pp. 1–12, 1959.