Cyclic Executives, Multi-Core Platforms and Mixed Criticality Applications

A. Burns
Department of Computer Science,
University of York, UK.
Email: alan.burns@york.ac.uk

T. Fleming
Department of Computer Science,
University of York, UK.
Email: tdf506@york.ac.uk

S. Baruah
Department of Computer Science,
University of North Carolina, US.
Email: baruah@cs.unc.edu

***Abstract*—Historically safety-critical real-time systems have been implemented using a cyclic executive (CE). Here a series of frames (minor cycles) are executed in sequence. Once the series is complete the sequence is repeated. The duration of the full sequence is often known as the major cycle. Within each frame, units of computation (jobs) are executed, again in sequence. Although there are a number of drawbacks to the use of CEs they have the advantage of being fully deterministic and efficiently implemented. For multi-core platforms, running a set of frames on each core is an obvious extension to the single core approach. Here there is advantage in coordinating the execution of the cores so that frames are released at the same time across all cores. For mixed criticality systems, the requirement for separation would imply that, at any time, code of the same criticality must execute on all cores. In this paper we consider how this requirement can be met and the performance, in terms of schedulability, it delivers. We consider partitioned and globally allocated work. For partitioned systems an allocation scheme is developed. For globally scheduled schemes we develop a polynomial-time sufficient schedulability test that determines whether a given mixed-criticality system is schedulable, and constructs a schedule if it is.**

I. Introduction

Two of the major challenges facing the developers of real-time systems are the widespread use of multi-core platforms and the increasing tendency for applications to contain components of different criticality. In this paper we consider these two challenges and focus on highly safety-critical application domains where cyclic executives are still the scheduling method of choice.

A cyclic executive is a simple deterministic scheme that consists, for a single processor, of the continuous executing of a series of *frames* (or *minor cycles* as they are often called). Each frame consists of a sequence of *jobs*. They execute in their defining sequence and must complete by the end of the frame. The set of frames is called the *major cycle* . So, for example, there might be 8 minor frames in the major frame, and each minor frame may be of 25ms duration. There are a number of drawbacks to using cyclic executives [1], [8] including

- Only periodic work is easily supported.
- All tasks must have a period that is a multiple of the minor cycle time; and a deadline no less than the minor cycle duration.
- Tasks can have a period that is at maximum the duration of the major cycle (unless secondary schedules are supported).
- Cyclic executes are not easy to construct or maintain (formally it is NP-hard in the strong sense to compute an optimal mapping of user tasks to the set of minor cycles).

Notwithstanding these restrictions, the run-time support needed for a cyclic executive is easy to implement and leads to efficient run-time behaviour. An application's schedulability is determined by construction – if the set of frames can be constructed then the application will meet all its deadlines.

On a multi-core, or multiprocessor, platform each core will have the same size of frame and the same major cycle time. For example, each of 4 cores could run with a minor frame of 25ms and a major frame of 200ms. The time source from which the run-time support software will execute the jobs contained within each frame, is synchronised so that each core is switching between minor cycles at the same time. Any cycle that is currently being executed is described as being *active*.

So on a multi-core platform with N cores there are, at any time t, zero or N active frames. The count is zero only if t represents the change over from one set of minor cycles to the next. Within each frame there are a series of jobs to be executed. If jobs are constrained to execute always within the same minor cycle and always on the same core then the run-time schedule is defined to be *partitioned*. Alternatively, if jobs can migrate from one active frame to another active frame on a different core then the schedule is defined to be *global*. In this paper we will look at both partitioned and global schemes.

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 a consideration was advocated by Giannopoulou et al. [10] 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 are concerned with a cyclic executive in which each minor cycle is partitioned into V criticality levels. Initially the highest criticality jobs are executed, when they have finished the next highest criticality jobs are executed. This continues until finally the lowest criticality jobs are executed. In a simple system with just two criticality levels, HI and LO, to support Giannopoulou's model, there is a switchover time S within each minor frame. Before S each core is executing HI-criticality work, after S each core is executing LO-criticality work. We refer to this scheme as *synchronised switching*, where each core has unrelated S values then the scheme is terms unsynchronised switching.

To give resilient fault tolerant behaviour, if the HI-criticality work has not completed by S on any core then the LO-criticality work is abandoned (on every core), thereby giving extra time for the HI-criticality work to execute (up to the end of the minor cycle). In this paper we will explore how to find acceptable (i.e safe and efficient) values for the switching times. We also investigate the loss of performance, in terms of schedulability, that results from the use of co-ordinated switching times (as apposed to having asynchronous switching which would not deliver the required separation).

A cyclic executive is a particularly restricted form of static schedule. The issue of mapping mixed criticality code to static schedules has been addressed by Tamas-Selicean and Pop [19], [21], [20] but without the constraint introduced by Giannopoulou et al.

As stated above, the Giannopoulou et al. constraint extends the applicability of MC-cognizant CPU scheduling to platforms including additional non-CPU resources. One of the objectives of the work reported in this paper is to determine the cost, in terms of schedulability loss, associated with enforcing the constraint.

An alternative approach to implementing the move between criticality levels in a static schedule is by switching between previously computed schedules; one per criticality level — this approach is explored by Baruah and Fohler [3]. Socci et al. [18] show how these Time-Triggered (TT) tables can be produced via first simulating the behaviour one would obtain from the equivalent fixed priority task execution. However, both of these prior schemes are focused on single processor systems.

Organisation. The remainder of this paper is organised as follows. In Section II, we elaborate upon the workload model that will be assumed in the remainder of this paper. We discuss partitioned cyclic executive scheduling of mixed criticality systems in Section III and global scheduling in Section IV. Conclusions are drawn together in Section V.

II. System Model

The cyclic executive (CE) is defined by two durations, T^F for the length of the minor cycle (frame) and T^M for the duration of the major cycle. These values are related by:

$$T^M = K.T^F$$

where K is a positive integer, and indeed is usually a power of 2. K is the number of frames in the repeating major cycle of the CE.

The issue of how to choose T^F and T^M to best support a set of tasks with given periods is beyond the scope of this paper. Rather we follow industrial practice [5] and assume these parameters are fixed by the system definition and that application tasks' periods are constrained to be multiples of T^F (up to the value of T^M).

The mapping of tasks to frames implies that there is a set of jobs allocated to each frame. All jobs within a frame must complete by the end of the frame. However, what it means to complete will depend on the behaviour of the system in terms of its criticality levels – as will be explained shortly.

We assume that the hardware platform consists of N identical (unit speed) processors (or cores). All jobs can execute on all cores and have identical temporal behaviour.

In general we assume there are V criticality levels, L_1 to L_V, with L_1 being the highest criticality[1]. Each job is assigned a criticality level. It also has two 'worst-case' computation times assigned. One represents its estimated execution time at its own criticality level ($C(L_i)$) and the other an estimate at the base criticality level ($C(L_V)$). It follows that if a job is of the lowest criticality level L_V then it only has one worst-case execution time. For all other jobs, $C(L_i) \geq C(L_V)$. The rationale for having more than one 'worst-case' execution time is covered in a number of papers on mixed criticality systems including the initial work of Vestal [22].

This use of only two C values for V criticality levels is a more constrained model than would result if each criticality level gave rise to a distinct computation time estimate. However with say five criticality levels it is unlikely that five distinct estimates of the worst-case execution time of the task would be available. The restriction to just two estimates is sufficient to capture the key properties of a mixed criticality system [17].

At run-time the system is defined to be executing in one of V modes. In mode L_V (the lowest) all deadlines of all jobs must be met. It represents 'normal' behaviour. If every job (τ_i) runs for no more than $C_i(L_V)$ then all deadlines must be guaranteed. If any job executes for more than $C_i(L_V)$ then the mode of the system will degrade towards L_1 in which only the highest criticality level jobs are guaranteed. This mode change behaviour is explained in more detail later in the paper.

Within a frame, jobs are executed in criticality order (highest first). A switch from one criticality level to the next (e.g. L_1 to L_2) is defined to occur at a specific time. This set of V-1 times (S^1 to S^{V-1}) is used to determine the schedulability of

[1]From a practical point of view V is unlikely to be greater than 5.

the CE; they may, or may not, be used to control switching at run-time – see Section III-F. As noted in the Introduction each core will switch between criticality levels at the same time.

III. Partitioned Cyclic Executives

In this section we look at the normal behaviour of a CE in which all jobs within a frame execute on the same core; i.e. the frame is statically allocated to a core. We require a means of deriving efficient values for the S parameters and a test for schedulability. We focus on the behaviour of a single frame, f, with, initially, just two criticality levels (HI and LO) and one switching time S. For the frame to be schedulable three properties must hold. In the LO-crit mode all HI-criticality jobs must complete by S and all LO-criticality jobs between S and the end of the frame at time T^F. In the HI-crit mode all HI-criticality jobs must complete by the end of the frame:

$$\sum_{k \in HI(f)} C_k(LO) \ \leq S \tag{1}$$

where $HI(f)$ is the set of HI-criticality jobs allocated to frame f,

$$\sum_{k \in LO(f)} C_k(LO) \ \leq T^F - S \tag{2}$$

where $LO(f)$ is the set of LO-criticality jobs allocated to frame f, and

$$\sum_{k \in HI(f)} C_k(HI) \ \leq T^F \tag{3}$$

Condition (1) is not just a test, it is a means of defining S. First, each concurrently executing frame f , compute the 'local' S (note there are N such frames, one on each of the cores):

$$S(f) \ = \ \sum_{k \in HI(f)} C_k(LO) \tag{4}$$

To give as much time as possible for the LO-criticality jobs to execute S should be as small as possible. But S must be the same on all concurrent frames (on the N cores). Given an allocation of jobs to frames, condition (4) can be used to compute the values $S(1)$ to $S(N)$. The coordinated switch time from HI to LO must therefore occur at time S_{max}:

$$S_{max} \ = \ \max_{f \in 1..N} S(f) \tag{5}$$

then condition (2) becomes:

$$\sum_{k \in LO(f)} C_k(LO) \ \leq T^F - S_{max} \tag{6}$$

The desire to maximise the time for LO-criticality work means that the allocation of jobs to frames must aim to minimise the maximum value of S. We shall return to this issue in Section III-B.

A. Multiple Criticality Levels

The above construction for two criticality levels can be easily extended to V levels, L_1 to L_V. In the following we overload the definition of the symbols L_i to also denote the set of jobs of that criticality. The V-1 switch points (S^1 to S^{V-1}) are constrained as follows (note we incorporate a value of S^0 which corresponds to the start of the frame (i.e $S^0 = 0$); so for each criticality level L_i:

$$\sum_{\tau_k \in L_i(f)} C_k(L_V) \ \leq S^i - S^{i-1} \tag{7}$$

where $L_i(f)$ is the set of L_i criticality jobs allocated to frame f; and

$$\sum_{\tau_k \in L_i(f)} C_k(L_i) \ \leq T^F - S^{i-1} \tag{8}$$

As with two criticality levels, the actual S values to use are the maximums of the values computed for each concurrent frame.

B. Maximising Schedulability

From consideration of the above analysis it is clear that a system is most likely to be schedulable if the switching times (the S values) are as early as possible. This means that the allocation of jobs to concurrent frames must aim to minimise the maximum value of each S^i value.

Determining a schedule of minimum duration for jobs allocated to a set of concurrent frames is equivalent to the bin-packing [13] problem, and is hence highly intractable: NP-hard in the strong sense. There are a number of techniques that can be applied to address the allocation problem. Hochbaum and Shmoys [11] 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 cores each of speed s, then the algorithm in [11] 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 [14]: the algorithm of [11] 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 [11]) is given augmented resources (in terms of faster cores) as compared to the resources available to the optimal algorithm.

Other applicable schemes come from formulating the allocation as an ILP problem [12], or the use of more general search techniques such as Genetic Algorithms [10], simulated annealing [7], [20] or Tabu-search [19], [21]. Here we apply efficient heuristics (First-Fit and Worst-Fit) as studied by Kelly et al [15]. The motivation being to demonstrate the effectiveness of choosing appropriate S values and to explore the schedulability loss resulting from having coordinated switching between criticality levels on each core.

In general the more effective one-pass bin packing algorithm is First-Fit with the items to be packed being dealt with in order of their size (biggest first). Here most effective means most likely to deliver an acceptable allocation. However, if the requirement is to have an even packing (as well as an acceptable one) then Worst-Fit is preferred. As we are attempting to minimise the maximum switching time from each core, an even allocation of work would seem an intuitive approach to investigate.

In the following investigations we restrict ourselves to considering the mapping of jobs of period equal to T^F to N cores. This ignores some of the issues of constructing CEs but retains the key problem of minimising the criticality switch time over the set of concurrently executing frames.

Our investigations take the common form of generating a large set of random job-sets and comparing the percentage of these job-sets that are schedulable. We increase the utilisation of the job-sets thereby increasing the difficulty in obtaining schedulability.

C. Evaluation Set-Up

The job-set parameters used in our experiments were randomly generated as follows:

- Job utilisations (U_i) were generated using the UUnifast algorithm [6], giving an unbiased distribution of utilisation values.
- Job periods were set at T^F (25ms in these experiments).
- Job deadlines were set equal to their periods (25ms).
- A job's criticality-aware estimate of worst-case execution time is given by: $C_i(L_i) = U_i/T_i$.
- For all but the lowest criticality level the 'base' estimate of execution time is given by: $C_i(L_V) = C_i(L_i) * CF$ – where the criticality factor CF is a random variable between X and Y.
- A four core platform was used for all evaluations.

Note CF is a random factor, if it were constant then the determination of the best allocation to minimise the switching times (the S values) would be more straightforward.

Two forms of experiments were undertaken. For each utilisation value (from 0 to 4 in steps of 0.05) 10,000 job-sets were generated.

In the first experiment only jobs of the highest criticality were considered. This was to investigate the estimation of the best switching time (S^1_{max} or just S_{max}). As indicated earlier, minimising this parameter is always the best possible result for the other criticality levels (as they must fit into the interval between S_{max} and T^R). Computing the other switching times is a repeat of this process.

In the second experiment two and four criticality levels are considered. Each criticality is allocated approximately the same utilisation. Switching times are computed and the job-set is considered schedulable if conditions 7 and 8 are satisfied for each criticality.

The motivation for this second experiment is to investigate how much performance lose (in terms of schedulability) results from having coordinated switching times across each core. The alternative (allow each core to switch when they are ready) does not have the property that the entire system is only ever executing code of the same criticality.

As we are focusing only on a single set of concurrently executing frames, there are four frames to allocate to (as their are four cores). In the following when we say we allocate a job to a core, we mean we allocate it to the frame executing on that core.

D. Evaluation Results – Experiment 1

In this experiment, which has only jobs of one criticality, the jobs are ordered biggest $C(L^1)$ to smallest. Each algorithm takes one job and tries to allocate it to a frame/core. To be an acceptable allocation the HI-criticality work must fit into the frame, so

$$\sum_{k \in L_1(f)} C_k(L_1) \ \leq T^F.$$

With First-Fit the job is allocated to the first frame that will accept it. With Worst-Fit the job is allocated to the frame that has the most free capacity (T^F - $C(L_1)$s of work already allocated to that frame).

Once all jobs are allocated (if this is possible) the S values of each frame are computed:

$$S(f) \ = \ \sum_{k \in L_1(f)} C_k(L_V)$$

and then the maximum value is obtained via equation (5).

Figure 1 shows the result of this experiment. As expected Worst-Fit works best (comparing top two lines). First-Fit will fill up the first core and therefore has a relatively high S_{max} value for a relatively low utilisation. Note this graph only contains schedulable job-sets.

But given the usual advantage of First-Fit we explored an alternative multi-pass scheme. On a single pass there is a fixed S (for all concurrent frames) and a job can only be allocated to a particular frame if, in addition to the constraint on the HI-criticality executions:

$$\sum_{k \in L_1(f)} C_k(L_V) \ \leq \ S(f).$$

If an acceptable allocation is found, then S is a safe value. Using a Branch-and-Bound approach the 'best' S value is computed. Here we bound possible S values to be the best and worst $S(f)$ values obtained from the standard First-Fit approach. In the usual way different S values are tried until no further improvements are possible. We term this scheme: First-Fit with Branch and Bound, FFBB.

The results of this FFBB approach are also contained in Fig. 1. They show that FFBB has a slight improvement over WF, especially at high levels of utilisation. Given that First-Fit (and therefore FFBB) scheduled more job-sets than Worst-Fit, we use FFBB in the second experiment. The superior behaviour of First-Fit over Worst-fit is demonstrated in the results for the second experiment (see below).

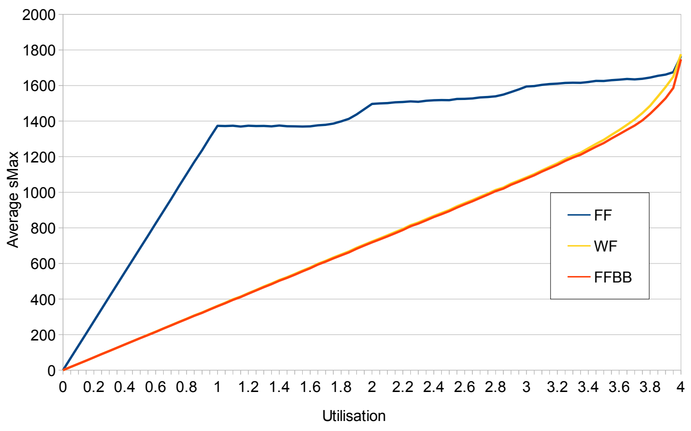

Fig. 1: Computing Easiest Possible Switch time

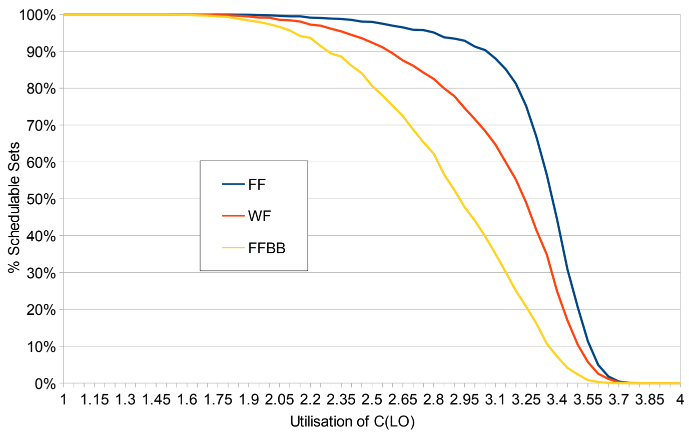

Fig. 2: Percentage of Schedulable Job-Sets (2 Crit-levels)

E. *Evaluation Results – Experiment 2*

The second experiment aims to demonstrate the application of the proposed scheme. First with two criticality levels and then four. The impact it has on schedulability is also evaluated.

The evaluation compares the results obtained from three allocations:

1) An allocation that does not compute or use the S_{max} value, but allocates jobs in criticality order (highest first, and largest computation time first within each criticality) using First-Fit– this is the scheme recommended by Kelly et al [15].
2) As above but using Worst-Fit.
3) The use of synchronised switching and FFBB on each criticality.

The first two schemes compute S values for each core but do not attempt to synchronise them (i.e. do not use S_{max}). They are more flexible and allow code of different criticality to run at the same time. It is inevitable that these two schemes will perform better than the more constrained synchronised switching (with FFBB).

The result of this experiment for two criticality levels is given in Fig. 2. Four criticality levels it is shown in Fig. 3.

The graphs in these two figures clearly show the degradation in performance (in terms of schedulability) that the synchronised switching suffers. For instance, with two criticality levels a utilisation of 3 (on a 4 core platform), synchronised switching will find only 40% of randomly generated job-sets to be schedulable. If switching is not synchronised then a figure close to 90% is witnessed. For four criticality levels the situation is even worse, almost no schedulable job-sets can be found with synchronised switching and utilisation of 3 – without synchronisation the rate is around 60%.

The cost of the approach is therefore high, but the separation that is delivered is of fundamental importance to mixed criticality systems. It is also possible to argue that a fairer comparison would be between synchronised switching with a collection of C values for the jobs, and unsynchronised switching with higher C values to reflect the other checks and

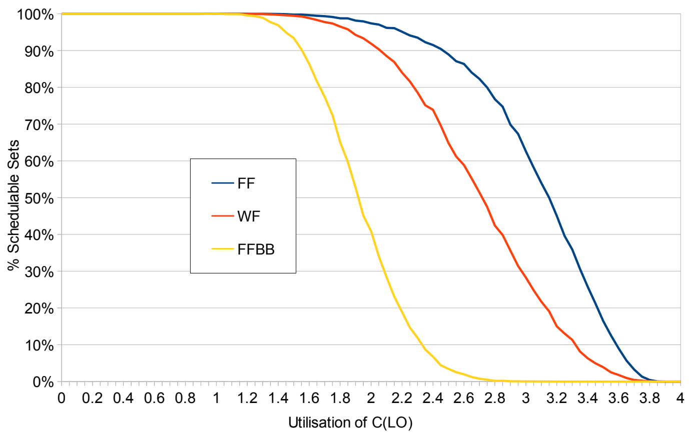

Fig. 3: Percentage of Schedulable Job-Sets (4 Crit levels)

assumptions that need to be made if code of different criticality execute at the same time. Looking again at the graph for 2 criticality levels. If one considers the 40% success rate, this is furnished at a utilisation of 3 for the synchronised scheme and 3.4 for the unsynchronised one. This difference may well be needed to give the same level of assurance.

F. *Run-Time Behaviour*

At run-time, interrupts at the times of the switching can be used to check that the necessary work has been completed. The necessary protocol could to be ether affirmative ("its OK to change") or negative ("do not change"):

- *affirmative*: each core broadcasts a message (or writes to shared memory) to say it has completed all its jobs of the current criticality jobs; when each core has completed its own work and has received (N-1) such messages it switches to the next criticality.
- *negative*: if any core is still executing application jobs at the switch time S it broadcasts a message to inform all other cores; any core that is not in receipt of such a message at time S+δ will move to its next set of jobs

(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 behaviour would result in no messages being sent; whereas the affirmative protocol would generate N broadcast messages. The affirmative protocol is, however, more resilient and less dependent on the temporal behaviour 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.

An efficient scheme, if directly supported by the hardware platform, and one supported by Giannopoulou et al. [10], is to use a synchronisation barrier. All cores call the barrier when they have completed L_1-criticality work, for example. When the final core completes, all calls are released from the barrier and each core knows it can continue with the next criticality level.

The benefit of this flexible scheme is that it can take advantage of the time gained by jobs executing for less than their worst-case. So at the end of the L_1 executions if the signal occurs before S^1 then all cores can move to L_2 executions early. Alternatively if it comes later, at time t relative to the start of the frame (with $S^1 < t$) then the run-time can decide that if there is sufficient time to start L_2 jobs then it should (i.e. if

$$\sum_{\tau_k \in L_2(f)} C_k(L_2) \ \leq T^F - t$$

then proceed).

There may indeed be situations arising at run-time when a late switch to one criticality is compensated by time gained from under-execution within the next set of jobs. So, for example, the switch to L_2 is late, but the switch to L_3 is not later then S^2 and hence the concurrent frames are able to undertake all their jobs.

At the end of the frame (T^F) there must be an interrupt to check that all jobs have completed. If not, work must be abandoned and the next set of concurrent frames must start on time (as they start with jobs of the highest criticality). The only exception to this rule would be when the old frame is still executing code of the highest criticality. This could only occur if $C(L_1)$ estimates for these jobs were optimistic – error recovery here would be application specific[2].

IV. GLOBAL SCHEDULING

We now address global scheduling where a job can start in one frame but complete in a different, concurrently executing, frame (on a different core). We restrict consideration here to just two criticality levels (HI and LO)[3]. As in the previous section we overload the meaning of the notation HI (and LO) to also imply the set of jobs of that criticality level.

The approach that we advocate here is again 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 [4] 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 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 the deadline for the jobs, 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 schedulability for the mixed-criticality instance is thus reduced to determining whether HI can be scheduled in such a manner that

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

In terms of the cyclic executive, Δ is the switch time S and D is the end of the frame, T^F.

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.)

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 [16, 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

$$\max\left(\frac{\sum_{i=1}^{n} c_i}{N}, \max_{i=1}^{n}\{c_i\}\right) \tag{9}$$

A direct application of McNaughton's result yields the conclusion that the minimum makespan Δ for a global preemptive

[2]One scheme used in industry is to extend the current minor cycle until all jobs have completed – the normal sequence of frame executing is then continued.

[3]An earlier version of the material presented in this section was contained in a workshop paper [2].

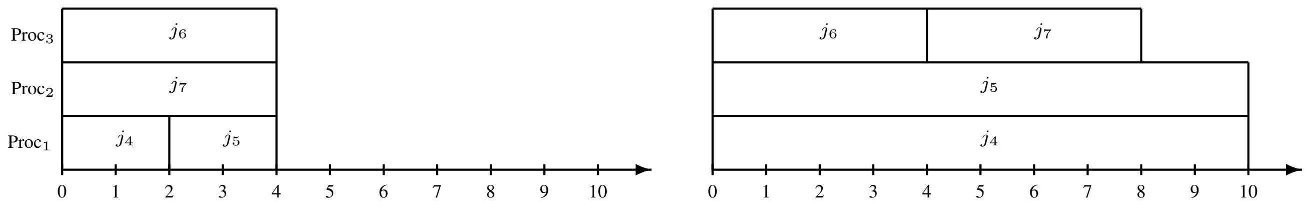

Fig. 4: Schedules for HI for the task system of Example 1. The left schedule is for a LO-criticality behavior, and has a makespan of four; it thus bears witness to the fact that this mixed-criticality instance satisfies Condition 11. The right schedule is for a HI-criticality behavior – it has a makespan of ten, thereby bearing witness that the instance satisfies Conditions 12 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.

schedule for the jobs in LO is given by

$$\Delta \overset{\text{def}}{=} \max\left(\frac{\sum_{LO} C(\text{LO})}{N}, \max_{LO}\{C(\text{LO})\}\right) \qquad (10)$$

Hence in any LO-criticality behavior it is necessary that the HI-criticality jobs must be scheduled to have a makespan no greater than $(D - \Delta)$:

$$\max\left(\frac{\sum_{HI} C(\text{LO})}{N}, \max_{HI}\{C(\text{LO})\}\right) \leq D - \Delta\ . \qquad (11)$$

(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 HI when each job executes for its HI-criticality WCET not exceed D:

$$\max\left(\frac{\sum_{HI} C(\text{HI})}{N}, \max_{HI}\{C(\text{HI})\}\right) \leq D, \qquad (12)$$

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

One might be tempted to conclude that the conjunction of Conditions 11 and 12 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 10, Δ evaluates to $\max(\frac{6+6+6}{3}, 6)$ or 6.
- The LHS of Condition 11 evaluates to $\max(\frac{2+2+4+4}{3}, 4)$, or 4. Condition 11 therefore evaluates to true.
- The LHS of Condition 12 evaluates to $\max(\frac{10+10+4+4}{3}, 10)$, or 10. Condition 12 therefore also evaluates to true.

However for this example, the schedule that causes Condition 11 to evaluate to true *must* have jobs j_6 and j_7 execute throughout the interval $[0, 4)$, while the one that causes Condition 12 to evaluate to true must have j_4 and j_5 execute throughout the interval $[0, 10)$ – see Figure 4. 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.

■

	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.

The example above illustrates that the conjunction of Conditions 11 and 12, with the value of Δ defined according to Equation 10, 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 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 [9].

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 10. 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 $(V_i \times V_{i+1} \times \mathcal{R}^+)$, $0 \leq i \leq 4$. The digraph G is thus a 6-layered graph – see Figure 5 – in which all edges connect vertices in adjacent layers. The sets of vertices in G are as follows:

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

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-criticality 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:

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

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 $\sum_{j_i \in HI} C_i(\textsc{hi})$. Achieving this flow will require that $C_i(\textsc{hi})$ units of flow go through $\langle 1, j_i\rangle$, which in turn requires that $C_i(\textsc{lo})$ units of flow go through $\langle 2, j_i, \textsc{lo}\rangle$, and $(C_i(\textsc{hi}) - C_i(\textsc{lo}))$ units of flow go through $\langle 2, j_i, \textsc{hi}\rangle$, for each $j_i \in HI$. This will require that all $C_i(\textsc{lo})$ units of flow from $\langle 2, j_i, \textsc{lo}\rangle$ go through $\langle 3, j_i, \alpha\rangle$; the flows from $\langle 2, j_i, \textsc{hi}\rangle$ through the vertices $\langle 3, j_i, \alpha\rangle$ and $\langle 3, j_i, \beta\rangle$ must sum to $(C_i(\textsc{hi}) - C_i(\textsc{lo}))$ units. The global schedule for 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 9) 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 theorem; we omit the details.

Theorem 1: If there is a flow of size

$$\sum_{j_i \in HI} C_i(\textsc{hi})$$

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 10 that $\Delta = 6$. The digraph constructed for this task system would require each of j_4–j_7 to transmit at least their corresponding $C_i(\textsc{lo})$'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 $(C_i(\textsc{hi}) - C_i(\textsc{lo}))$ 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

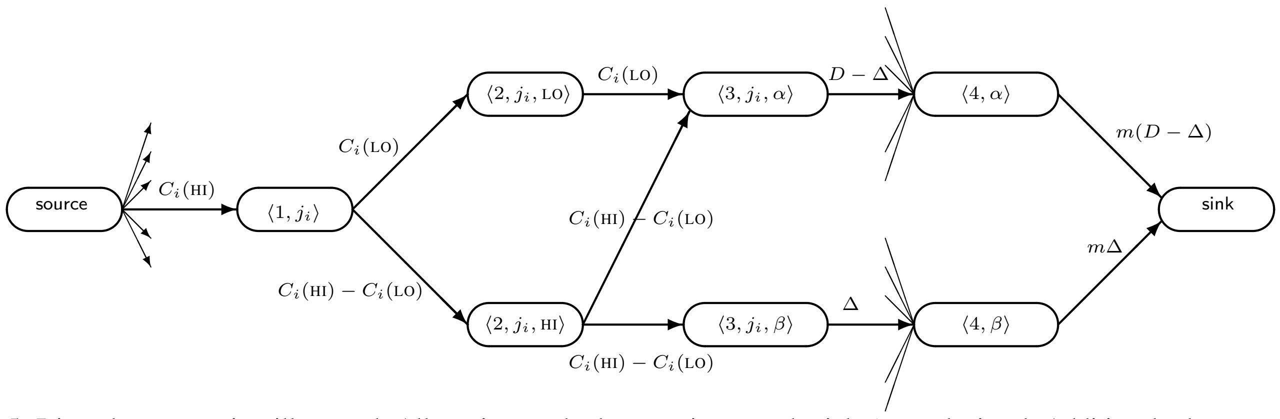
source
$C_i(\text{HI})$
$\langle 1, j_i \rangle$
$C_i(\text{LO})$
$C_i(\text{HI}) - C_i(\text{LO})$
$\langle 2, j_i, \text{LO} \rangle$
$\langle 2, j_i, \text{HI} \rangle$
$C_i(\text{LO})$
$C_i(\text{HI}) - C_i(\text{LO})$
$C_i(\text{HI}) - C_i(\text{LO})$
$\langle 3, j_i, \alpha \rangle$
$\langle 3,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

Fig. 5: Digraph construction illustrated. All vertices and edges pertinent to the job j_i are depicted. Additional edges emanate from vertex source to a vertex $\langle 1, j_\ell \rangle$, for each $j_\ell \in HI$; 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 $j_\ell \in HI$.

of size $\sum_{j_i \in HI\}} C_i(\text{HI})$, 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 [9], 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.

As also noted earlier, the above formulation in terms of the parameters D and Δ is directly applicable to a simple frame based cyclic executive where $D = T^F$ and $\Delta = S$. However, the extension of this work on global scheduling to multiple criticality levels forms part of further work, although we do not see any fundamental problems with such an extension.

V. Conclusions

Mixed-criticality scheduling theory must extend consideration beyond simply CPU computational demand, as characterised 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. [10] – 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 cyclic executives (CEs) that implement this approach. Each frame of the CE consists of a collection of independent jobs that share a common release time and deadline. Across the N cores of our multiprocessor platform N frames execute in parallel and share a common release time and duration.

Our basic scheme packs each frame with jobs in criticality order (highest criticality first, lowest criticality last). Between each criticality the whole system switches in a synchronised way to the ensure only jobs of the same criticality execute concurrently.

Mixed criticality schedulability analysis is used to determine the worst-case switching times – and thereby determine schedulability. At run-time a barrier-based synchronisation primitive can be used to coordinate the switches when the preconditions are met (all jobs of the 'old' criticality have completed).

Two different allocation methods are considered in this paper. Partitioned, where job only execute on one core in one frame; and global where jobs may complete in a different (though concurrently executing) frame on another core. We observed that:

- 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 investigated the use of common heuristics (Best-Fit and Worst-Fit) to effectively map application tasks on to the multi-core CE.
- 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.

Using an evaluation based on randomly generated set of jobs we were able to estimate the reduction in schedulability that arises from the requirement that only code of the same criticality executes at the same time. The design of a mixed criticality system must decide how, in a disciplined way, to reconcile the conflicting requirements of partitioning/separation for (safety) assurance and sharing for efficient resource usage. We have been able to address this issue for the particular approach towards mixed-criticality scheduling advocated in [10], and to the use of Cyclic Executives for multi-core platforms.

Acknowledgements

This research is partially supported by NSF grants CNS 1115284, CNS 1218693, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors Corp. It is also supported by ESPRC grant MCC (EP/K011626/1). No new primary data were created during this study.

References

[1] T. P. Baker and A. Shaw. The cyclic executive model and ada. In *Proc. 9th IEEE Real Time Systems Symposium*, pages 120–129, 1988.

[2] S. Baruah and A. Burns. Achieving temporal isolation in multiprocessor mixed-criticality systems. In L. Cucu-Grosjean and R. Davis, editors, *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 21–26, 2014.

[3] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proc. of IEEE Real-time Systems Symposium 2011*, December 2011.

[4] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proc. of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 13–22. IEEE, 2010.

[5] I. Bate and A. Burns. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Systems Journal*, 25(1):5–37, 2003.

[6] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Journal of Real-Time Systems*, 30(1-2):129–154, 2005.

[7] A. Burns, N. Hayes, and M. Richardson. Generating feasible cyclic schedules. *Control Engineering Practice*, 3(2):151 – 162, 1995.

[8] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 4th edition, 2009.

[9] L. Ford and D. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:144–162, 1956.

[10] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proc. Int. Conference on Embedded Software (EMSOFT)*, Montreal, 2013.

[11] D. Hochbaum and D. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[12] M. Jan, L. Zaourar, V. Legout, and L. Pautet. Handling criticality mode change in time-triggered systems through linear programming. *Ada User Journal, Proc of Workshop on Mixed Criticality for Industrial Systems (WMCIS'2014)*, 35(2):138–143, 2014.

[13] D. Johnson. *Near-Optimal Bin-Packing Algorithms*. PhD thesis, Department of Mathematics, MIT, 1974.

[14] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM (JACM)*, 47(4):617–643, 2000.

[15] O. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1051–1059, 2011.

[16] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[17] D. Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium*, pages 291–300. IEEE Computer Society, 2009.

[18] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed critical scheduler. In *Proc. WMC, RTSS*, pages 67–72, 2013.

[19] D. Tamas-Selicean and P. Pop. Design optimisation of mixed criticality real-time applications on cost-constrained partitioned architectures. In *Real-Time Systems Symposium (RTSS)*, pages 24–33, 2011.

[20] D. Tamas-Selicean and P. Pop. Optimization of time-partitions for mixed criticality real-time distributed embedded systems. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 2–10, 2011.

[21] D. Tamas-Selicean and P. Pop. Task mapping and partition allocation for mixed criticality real-time systems. In *IEEE Pacific Rim Int. Sym. on Dependable Computing*, pages 282–283, 2011.

[22] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.