# Migrating Mixed Criticality Tasks within a Cyclic Executive Framework

Alan Burns[1] and Sanjoy Baruah[2]

[1] University of York, UK.
[2] University of North Carolina, US.

**Abstract.** In a cyclic executive, a series of frames are executed in sequence; once the series is complete the sequence is repeated. Within each frame, units of computation are executed, again in sequence. In implementing cyclic executives upon multi-core platforms, 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 additionally require that, at any time, code of the same criticality should be executing on all cores. In this paper we derive algorithms for constructing such multiprocessor cyclic executives for systems of periodic tasks, when inter-processor migration is permitted.

## 1 Introduction

Recent trends in embedded computing towards the widespread use of multi-core platforms, and the increasing tendency for applications to contain components of different criticality, have thrown up major challenges to the developers of reliable software-based systems. In this paper we consider these two challenges in the context of highly safety-critical application domains where cyclic executives remain the scheduling mechanism of choice.

**Cyclic executives.** 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* that execute in the specified sequence and are required to complete by the end of the frame. The set of frames is called the *major cycle*.

**Multicore CPUs.** On a multi-core, or multiprocessor, platform each core should have the same frame size and the same major cycle time. The time source from which the run-time support software will execute the jobs contained within each frame, is synchronised so that all cores switch between minor cycles concurrently. 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 allow a small number of constrained job migrations. In a previous workshop paper [5] we focused on independent jobs, in this paper we address the more practical problem

of jobs that are derived from periodic tasks. In other work [6, 7] we have shown how fully partitioned systems can be constructed.

**Mixed criticality.** In mixed-criticality scheduling (MCS) theory, tasks are characterized by several different WCET parameters denoting different estimates of the true WCET value, these different estimates being made at different levels of assurance. The scheduling objective is then to validate the correct execution of each task at a level of assurance that is consistent with the criticality level assigned to that task: tasks assigned greater criticality must be shown to execute correctly when more conservative WCET estimates are assumed, while less critical tasks need to have their correctness demonstrated only when less conservative WCET estimates are assumed.

**Related work.** 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 [13, 14]. 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 in [3, 12]. However, these schemes are only applicable to single processor systems. The notion of separation used in this paper comes from [9].

## 2 System Model

In a typical implementation, a cyclic executive (CE) is defined by two durations, the length of the minor cycle (or frame) $T_F$ and the duration of the major cycle $T_M$. These values are related by ($T_M = k.T_F$) where $k$ is a positive integer (usually a power of 2), denoting 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 [4] 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 $m$ identical (unit speed) processors (or cores). Each job can execute on any core and has identical temporal behaviour on all cores.

In general $V$ criticality levels, $L_1$ to $L_V$, may be defined for a system with $L_1$ being the highest criticality; in this paper we primarily restrict ourselves to just two criticality levels ($V = 2$), and use the notation $L_1 = \text{HI}$ and $L_2 = \text{LO}$.

**Run-time support**

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.

An interesting approach towards such a consideration was advocated by Giannopoulou et al. [9] in the context of multicore platforms: during any given instant in time, all the cores are only allowed to execute code of the same criticality level. This approach has the advantage of ensuring that accesses to all shared resources (memory buses, cache, etc.) during any time-instant are only from code of the same criticality level. We refer to such a scheme of switching between workloads of different criticality levels as *synchronised switching*.

We focus our attention in this paper on synchronized switching. That is, we seek to construct cyclic executives in which each minor cycle may be considered partitioned into $V$ criticality levels. Initially the highest criticality jobs are executed, when they have finished the next highest criticality jobs are executed, and so on. This continues until finally the lowest criticality jobs are executed. In a simple system with just two criticality levels, HI and LO, there is a switchover time $S$ defined within each minor frame. Before $S$ each core is executing HI-criticality work, after $S$ each core is executing LO-criticality work. To give resilient fault tolerant behaviour, if the HI-criticality work has not completed by time-instant $S$ on any core then the LO-criticality work is postponed (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 (safe and efficient) values for the switching times.

**Implementing the criticality switches.** Giannopoulou et al. [9] advocated, if supported by the hardware platform, the use of synchronisation barriers. In the case of dual-criticality workloads (the generalization to $> 2$ criticality levels is straight-forward), each core calls the barrier upon completing its assigned HI-criticality work. When the final core completes and calls the barrier, all the calls are released from the barrier and each core continues with executing LO-criticality work.

The benefit of this barrier-based scheme is that it can take advantage of time gained by jobs executing for less than their estimated WCETs. So at the end of the HI-criticality executions if the signal occurs before the pre-computed barrier $S$, then all cores can move to LO-criticality executions early. Additionally, there may be situations arising at run-time when a late switch to one criticality level is compensated by time gained from under-execution within jobs of the next criticality level. For example, the switch occurs at some time $> S$, but the LO-criticality jobs end up executing for less than their LO-criticality WCET values and hence all complete by the end of the frame.

## 3 Dual Criticality Jobs

In this section, we consider the scheduling of a collection of jobs within a single frame of an $m$-processor platform, when there are only two criticality levels ($V = 2$). All the jobs are assumed to become available at the start of the frame (without loss of generality, denoted as being at time 0), and they all have a deadline at the end of the frame (denoted $D$). In keeping with prior work on the scheduling of such dual-criticality systems, we use the notation HI and LO to denote the greater and lesser criticality levels (i.e., $L_1 = \text{HI}$ and $L_V = L_2 = \text{LO}$). The criticality of job $j_i$ is denoted by $\chi_i \in \{\text{LO}, \text{HI}\}$; each HI-criticality job is characterized by two WCET parameters $C_i(\text{LO})$ and $C_i(\text{HI})$ (with $C_i(\text{LO}) \leq C_i(\text{HI})$), while each LO-criticality job $j_i$ is characterized by a single WCET parameter $C_i(\text{LO})$ (for convenience such jobs are also assigned a $C_i(\text{HI})$ value with $C_i(\text{LO}) = C_i(\text{HI})$).

Given a collection of such dual-criticality jobs to be scheduled within a frame of duration $D$ upon an $m$-processor platform, our objective is to determine the switching point $S$ such that only HI-criticality jobs are executed over the interval $[0, S)$. If all HI-criticality jobs complete by time-instant $S$, then LO-criticality jobs are executed over $[S, D)$; else, the LO-criticality jobs are abandoned and execution of HI-criticality jobs continues over $[S, D)$ as well. It follows that there are three conditions that need to be satisfied:

1. If each HI-criticality job $j_i$ executes for no more than $C_i(\text{LO})$, then all the HI-criticality jobs must fit into the interval $[0, S)$.
2. All the LO-criticality jobs must fit into the interval $[S, D)$
3. If each HI-criticality job $j_i$ executes for no more than $C_i(\text{HI})$, then all the HI-criticality jobs must fit into the interval $[0, D)$.

In Section 3.1 below, we derive a simple and efficient algorithm for determining $S$ (and the corresponding schedules) such that these conditions are satisfied; in Section 3.2, we describe an optimization to this simple method. These algorithms assume minimal run-time support.

### 3.1 A simple scheme for constructing CEs

We first define two (potential) candidates for the switching point $S$:

$S^{\min}$ The earliest instant at which all HI-criticality jobs have completed if they execute for no more than $C(\text{LO})$.

$S^{\max}$ The latest instant at which a switch must occur for the LO-criticality work to complete by time $D$.

It is evident that any candidate $S$ must satisfy the two inequalities $S^{\min} \leq S \leq S^{\max}$.

Let us additionally define two interval durations, which constrain the possible values of $S^{\min}$ and $S^{\max}$.

$\Delta^{\text{LO}}$ The duration (makespan) of the interval needed for all the LO-criticality jobs to (begin and) complete execution.

$\Delta^{\text{HI}}$ The duration of the interval needed for all the HI-criticality jobs to execute the extra work they must do in HI-criticality mode — i.e., the amount $(C_i(\text{HI}) - C_i(\text{LO}))$, for each $j_i$ with $\chi_i = \text{HI}$.

To determine these durations, we employ the optimal scheme of McNaughton [10, page 6]. Given a collection of $n$ jobs with execution requirements $c_1, c_2, \ldots, c_n$, McNaughton showed that the minimum makespan of a preemptive schedule for these jobs on $m$ unit-speed processors is given by

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

The actual schedule is obtained by taking the jobs (in any order) and allocating them to $m$ intervals of the size of the makespan, each representing one of the $m$ processors. As one interval is filled, perhaps with part of a job, the next interval starts with the rest of this job. At most $(m-1)$ jobs are split across intervals in this manner. During run-time a job that was split across two intervals will run at the beginning of the time-interval upon one processor, and towards the end of the time-interval on the other processor.

A direct application of McNaughton's result yields the conclusion that the minimum makespan for a global preemptive schedule for the jobs in LO-criticality mode is given by

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

We therefore set

$$S^{\max} \stackrel{\text{def}}{=} D - \Delta^{\text{LO}} \tag{3}$$

Similarly, a direct application of the makespan result allows the minimum interval for the HI-criticality work (in LO-criticality mode) to be computed:

$$S^{\min} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{LO})}{m}, \max_{\chi_i=\text{HI}}\left\{C_i(\text{LO})\right\}\right) \tag{4}$$

Clearly for the whole system to be schedulable, it is necessary that $S^{\min} \leq S^{\max}$ which is equivalent to requiring that

$$\begin{aligned} & S^{\min} \leq D - \Delta^{\text{LO}} \\ \Leftrightarrow\ & S^{\min} + \Delta^{\text{LO}} \leq D \end{aligned} \tag{5}$$

We now consider the final constraint — the scheduling of HI-criticality jobs executing in HI-criticality mode. It has been shown [2, Example 1] that this is not necessarily ensured by simply computing the makespan (using McNaughton's method, as above) with the $C_i(\text{HI})$ values, and validating that the resulting makespan is $\leq D$. We instead determine the minimal makespan for all the HI-criticality jobs, subject to each such job having received an amount of execution equal to its LO-criticality WCET by time-instant $S^{\min}$. To determine this

makespan, we apply McNaughton's scheme to the work that is left to do after time-instant $S^{\min}$ (i.e. $C_i(\text{HI}) - C_i(\text{LO})$ for each job $j_i$ with $\chi_i = \text{HI}$). Letting $C_i(\text{EX})$ denote the "excess" computational requirement of job $j_i$ in HI-criticality mode over LO-criticality mode:

$$C_i(\text{EX}) \stackrel{\text{def}}{=} \big(C_i(\text{HI}) - C_i(\text{LO})\big),$$

we have

$$\Delta^{\text{HI}} \stackrel{\text{def}}{=} \max\left(\frac{\sum_{\chi_i=\text{HI}} C_i(\text{EX})}{m}, \max_{\chi_i=\text{HI}}\big\{C_i(\text{EX})\big\}\right) \tag{6}$$

It is evident that $S^{\min} + \Delta^{\text{HI}} \leq D$ is sufficient for schedulability; earlier (Expression 5) we had shown that $S^{\min} + \Delta^{\text{LO}}$ should also be $\leq D$. Putting these pieces together, we may summarize this method as follows. We compute $S^{\min}, \Delta^{\text{LO}}$, and $\Delta^{\text{HI}}$ according to Expressions (4), (2), and (6) respectively, and require that

$$S^{\min} + \max\big(\Delta^{\text{LO}}, \Delta^{\text{HI}}\big) \leq D \tag{7}$$

as a sufficient schedulability condition. If this condition is satisfied, $S \leftarrow S^{\min}$ (i.e., we declare $S^{\min}$ to be the switch-point we had set out to compute).

### 3.2 An improvement

Let us now suppose that Condition 7 is violated, and $S^{\min} + \max\big(\Delta^{\text{LO}}, \Delta^{\text{HI}}\big) > D$. Since $\big(S^{\min} + \Delta^{\text{LO}} \leq D\big)$ is a necessary condition for schedulability (see Inequality 5), it must be the case that

$$S^{\min} + \Delta^{\text{HI}} > D.$$

Now if $\big(\sum_{\chi_i=\text{HI}} C_i(\text{HI}) \geq mD\big)$, there is nothing to be done. Otherwise, there must be some unused processor capacity in the McNaughton schedule constructed according to Expression 4 for the interval $[0, S)$, and/or in the McNaughton schedule constructed according to Expression 6 for the interval after time-instant $S$. Let us consider the situation where the schedule has some unused processor capacity over the interval $[0, S)$ (recall that $S \leftarrow S^{\min}$ in the method of Section 3.1). An inspection of Expression (4) reveals that this happens if

$$\frac{\sum_{\chi_i=\text{HI}} C_i(\text{LO})}{m} < \max_{\chi_i=\text{HI}}\big\{C_i(\text{LO})\big\}$$

Our idea, intuitively speaking, is that any such unused capacity prior to time-instant $S$ may as well be allocated to some HI-criticality task, for use in the event of the system undergoing a mode-change into HI-criticality mode. (If the system does not undergo such a mode-change, this allocated capacity may end up remaining unused.) Doing so leaves less execution remaining to be completed after the switch instant $S$ in HI-criticality mode, and may thus result in a smaller makespan in HI-criticality modes (i.e., a smaller value for $\Delta^{\text{HI}}$).

Such a scheme is particularly effective if the duration of the HI-criticality schedule after $S$ — the one of duration $\Delta^{\text{HI}}$ — is also dominated by longer jobs, i.e., if in Expression 6

$$\frac{\sum_{\chi_i=\text{HI}} C_i(\text{EX})}{m} < \max_{\chi_i=\text{HI}} \{C_i(\text{EX})\}$$

If this be the case, then the unused capacity prior to time-instant $S$ can be filled so as to minimise the maximum $C_i(\text{EX})$ by bringing forward work to before $S$ — this is accomplished by increasing $C_i(\text{LO})$ for such a job, thereby decreasing its $C_i(\text{EX})$ by the same amount. However, jobs that have ($C_i(\text{LO}) = S$) cannot have work brought forward in this manner since this would result in $S$ increasing as well.

It is evident that this scheme is effective since:

- Any work brought forward will not change $S$,
- The first term in Expression (6) is not increased by bringing work forward, and
- The second term in Expression (6) is reduced by always choosing the largest value and decreasing it.

We note that if more than one job has the same $C_i(\text{EX})$ value then an arbitrary choice is made (and has no impact on optimality).

And what if there is no unused processor capacity in the schedule over $[0, S)$? In that case, the switch-point $S$ may be increased to any value $\leq S^{\max}$ (where $S^{\max}$ is as defined by Expression (3)). An obvious choice for $S$ is $S \leftarrow S^{\max}$; an iterative algorithm for achieving the smallest value of $S$ (i.e., the earliest possible switch-time) is as follows. Setting the switch point $S$ to be $S^{\min} + 1$ will generate $m$ free slots. So $C_i(\text{LO})$ values of HI-criticality jobs can be increased by this amount (and the corresponding $C(\text{EX})$ values decreased). If this will reduce the size of $\Delta^{\text{HI}}$ by more than one then an overall decrease in $S + \Delta^{\text{HI}}$ will have been achieved. This cycle is repeated (i.e. adding 1 to $S$) until either no further gain is made or $S$ takes the value of $S^{\max}$. At each step of the cycle no $C(\text{LO})$ value should increase beyond the current value of $S$.

**Example 1** We apply this improved scheme to the scheduling of the mixed-criticality instance of Table 1 upon 3 unit-speed processors with a frame length of 8 ($D = 8$).

We can immediately use the equations above to compute: $\Delta^{\text{LO}} = 3$ (and hence $S^{\max} = 5$) and $S^{\min} = 4$. So the first step to schedulability is satisfied (i.e. $S^{\min} \leq S^{\max}$). if we ignore mixed criticality issues then the minimum makespan for the HI-criticality jobs (ignoring LO-criticality work) is 7. So a completely separated scheme would require a frame size of 10 ($7 + 3$).

If we initially focus on $S^{\min}$ then we note that there are no free slots, so equation(6) gives a makespan in HI-criticality mode ($\Delta^{\text{HI}}$) of 5. So the use of this value for $S$ (i.e. 4) gives a required frame size of 9 (4+5); since the frame-size is 8, the instance would be deemed unschedulable with $S \leftarrow 4$.

| | $\chi_i$ | $C_i(\text{LO})$ | $C_i(\text{HI})$ | $C_i(\text{HI}) - C_i(\text{LO})$ |
|---|---|---|---|---|
| $j_1$ | LO | 3 | - | - |
| $j_2$ | LO | 2 | - | - |
| $j_3$ | LO | 2 | - | - |
| $j_4$ | HI | 2 | 7 | 5 |
| $j_5$ | HI | 3 | 7 | 4 |
| $j_6$ | HI | 3 | 3 | 0 |
| $j_7$ | HI | 4 | 4 | 0 |

**Table 1.** An example dual-criticality job instance

However, if we set $S \leftarrow (S^{\min}+1)$ which equals $S^{\max} = 5$ then the total work available on three processors by time 5 is 15. The work required using $C(\text{LO})$ values for HI-criticality work is 12. Hence 3 units of work can be added to these $C(\text{LO})$ values. If we make $C_4(\text{LO}) = 4$ and $C_5(\text{LO}) = 4$ then maximum $C_i(\text{EX})$ becomes equal to 3. Hence $\Delta^{\text{HI}} = 3$ and $S^{\max} + \Delta^{\text{HI}} = 8$. Therefore the job set fits into the frame size of 8, with a switch time of 5. □

Rather than iterating through potential candidate values for $S$ in the manner described above, we can construct a single **linear program** (LP) for determining a suitable value for $S$ – see Figure 1. In this linear program

- $\delta_i$ denotes the amount of execution that is "moved" from $C_i(\text{EX})$ to $C_i(\text{LO})$; the first two constraints of the LP restrict this amount to (i) be positive and (ii) not exceed the value of $C_i(\text{EX})$.
- The next two constraints are an LP representation of the makespan resulting from applying McNaughton's rule to the LO-criticality execution requirements of the HI-criticality jobs.
- The fifth constraint represents the requirement that the synchronization barrier should not be moved beyond $S^{\max}$ (since doing so could result in LO-criticality jobs failing to complete even in LO-criticality behaviors).
- The final two constraints are an LP representation of the makespan resulting from applying McNaughton's rule to the excess (i.e., HI-criticality minus LO-criticality) execution requirements of the HI-criticality jobs.

Since a linear program can be solved in time polynomial in its representation, this LP-based approach allows us to determine, in polynomial time, whether an instance can be scheduled using our improved approach. (The iterative approach could require time proportional to $S^{\max} - S^{\min}$; in pathological cases, it could thus have a run-time that is pseudo-polynomial in the representation of the instance to be scheduled.)

## 4 Periodic Task Systems

We now consider instances in which the workload is specified as periodic tasks rather than as individual jobs. In this section, we again focus upon dual-criticality systems.

MINIMIZE $(S + S')$ subject to

$$
\begin{aligned}
&(\mathbf{1}).\ \ \delta_i \geq 0 && \text{for each } i \ : \ \chi_i = \text{HI} \\
&(\mathbf{2}).\ \ \delta_i \leq C_i(\text{EX}) && \text{for each } i \ : \ \chi_i = \text{HI} \\
&(\mathbf{3}).\ \ S \geq C_i(\text{LO}) + \delta_i && \text{for each } i \ : \ \chi_i = \text{HI} \\
&(\mathbf{4}).\ \ S \geq \Big( \sum_{i:\chi_i=\text{HI}} (C_i(\text{LO}) + \delta_i) \Big) / m \\
&(\mathbf{5}).\ \ S \leq S^{\max} \\
&(\mathbf{6}).\ \ S' \geq C_i(\text{EX}) - \delta_i && \text{for each } i \ : \ \chi_i = \text{HI} \\
&(\mathbf{7}).\ \ S' \geq \Big( \sum_{i:\chi_i=\text{HI}} (C_i(\text{EX}) - \delta_i) \Big) / m
\end{aligned}
$$

**Fig. 1.** A linear program for determining the switching point $S$

Let us assume that there are $k$ minor cycles in the major cycle of the cyclic executive we seek to construct, with $k$ being an integer power of 2. As before, we assume that we have $m$ parallel cores. Application tasks are assumed to have *harmonic* periods that are $k$ or $2k$ or $4k$ etc. times the size of the minor frame (e.g. $\in \{25ms, 50ms, 100ms, 200ms, 400ms, \ldots\}$). For the purposes of illustration in the following discussions we will assume that there are 8 minor cycles to the major cycle (i.e., $k = 8$).

We now describe how we construct cyclic executives for such dual-criticality periodic task systems. For each of the $k$ sets of frames we seek to compute a switch point $S_1, S_2, \ldots, S_k$; we do not require these switch points to be the same in each minor cycle.

First the tasks with period equal to the minor cycle must be allocated to all the minor cycles. These can be dealt with by the job-based procedures described in Section 3.

To add tasks with longer periods a number of approaches are possible. The simplest, and the one that is most appropriate if computation times for these tasks are relatively small (and hence compatible with the jobs already allocated), is to allocate each job of these tasks to exactly one minor cycle. So, for example, a task with period equal to twice the minor cycle duration will be allocated exactly once each in cycles $\{1, 2\}$, $\{3, 4\}$, $\{5, 6\}$, and $\{7, 8\}$. And a task with period four times the minor cycle duration will be allocated exactly once each in cycles $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8\}$. Finally tasks with period equal to the major cycle can be allocated to any one of the minor cycles. To manage this allocation, common forms of heuristics may be applied. First-Fit or Worst-Fit for example, with the tasks been allocated largest $C(\text{LO})$ first. As tasks are added to each cycle the analyses of Section 3 above for the set of frames that make up that cycle are applied. Different switch points for each cycle will emerge, but if the full task set can be accommodated then allocation is complete and the system is schedulable by construction.

This process of allocating jobs to single cycles can fail if tasks with larger periods have larger computation times ($C(\text{LO})$ or $C(\text{HI})$) that are not easily accommodated within a single frame. To accommodate such tasks, jobs need to be split between minor frames. (This is a common approach with single processor cyclic executives and is considered to be one of the disadvantages of the cyclic executive approach.) Two forms of splitting are possible, *explicit* or *implicit*. With explicit splitting the code of the task is actually partitioned (statically). So for a task with period equal to twice the minor cycle the code will be 'cut' in half (approximately). Each portion can then be analysed to determine its $C(\text{LO})$ and $C(\text{HI})$ values. The first half will be allocated to cycles 1, 3, 5 and 7; and the second half will be allocated to cycles 2, 4, 6 and 8. These jobs are added to the existing jobs corresponding to tasks with periods equal to the duration of a single frame, and the earlier analysis of Section 3 again applied.

Although this explicit splitting is optimal from a scheduling point of view (if the code can be partitioned exactly into two parts with the same $C(\text{LO})$ and $C(\text{HI})$ values), this approach suffers from a number of significant practical problems:

- The lack of available tool support for splitting code into exact portions that can give rise to identical estimates of worst-case execution time.
- Code structures may not be amenable to such partitioning.
- Even if approximate splitting is possible, modifications to the code due to upgrades or bug fixes, will require re-splitting, and re-testing. This is an expensive process.

For these reasons we reject explicit splitting and employ an implicit scheme similar to that used earlier for job splitting. But for a task with two estimates of worst-case execution time there is the issue of when to trigger the migration. (Note the migration here is to the next cycle; it may or may not involve a move to a different core.)

Reducing a task that runs every 50ms, say, to one that runs every 25ms is very similar to the use of period transformation [11] to reduce a task's period (and hence raise its priority in a rate-monotonic system). The application of period transformation to mixed criticality systems has been discussed in a number of papers [1, 8, 15]. Here we make use of the main techniques which is to divide $C(\text{HI})$ by the number of parts the task is split into. So if the 50ms task has WCET estimates of 8 and 12 and is split into two parts, its computation time in the first cycle will be all at the "normal" or LO-criticality level (so it has $C(\text{LO}) = 6$ and $C(\text{EX}) = 0$). In the second cycle it could again have $C(\text{LO}) = 6$ and $C(\text{EX}) = 0$, but this would be conservative in that the task is being allocated $6 + 6 = 12$ units even at the LO-criticality level – it would be more efficient to have $C(\text{LO}) = 2$ and $C(\text{EX}) = 4$. Moreover, if the LO-criticality load on the first cycle is too high it could reduce its requirement in that cycle to be $C(\text{LO}) = 5$, and then in the second cycle we have $C(\text{LO}) = 3$ and $C(\text{EX}) = 4$. Alternatively if the second phase of the cycle is overloaded in the second cycle (i.e. $C(\text{EX}) = 4$ is too high) then the first cycle could have $C(\text{LO}) = 8$ and $C(\text{EX}) = 2$, and the

second cycle $C(\text{LO}) = 0$ and $C(\text{EX}) = 2$. This potential movement of work from one cycle to another is exploited in the following scheme.

We need additional notation to denote per-cycle parameterisation. We will add '$[x]$' to the previously defined terms to denote the $x$th cycle. A task, $\tau_i$, with a period equal to $k$ minor cycles is split into $k$ jobs, $\tau_i[1] \ldots \tau_i[k]$. Its computation times will be denoted by $C_i[x](\text{LO})$, $C_i[x](\text{HI})$ and, by construction, $C_i[x](\text{EX}) \stackrel{\text{def}}{=} C_i[x](\text{HI}) - C_i[x](\text{LO})$.

We now describe the allocation process for dual critically systems. The following steps will be undertaken:

1. Allocate all single cycle tasks (i.e tasks with period equal to the minor cycle) using the job-based analysis developed earlier (discussed in Section 3 above).
2. Allocate all remaining HI-crit tasks using the period transformation scheme (as detailed below).
3. If the above step is not successful, move work to later cycles until all HI-crit work is scheduled (or declare task set is unschedulable)
4. Allocate all remaining LO-crit tasks.

We will now describe these steps in more detail.

Initially the HI-crit tasks are allocated to the cycles with the computation time of each part of the task $\tau_i$ being defined by:

$$C_i[x](\text{LO}) = \min\left(\frac{C_i(\text{HI})}{p}, C_i(\text{LO}) - \frac{(x-1)C_i(\text{HI})}{p}\right)_{\geq 0} \tag{8}$$

(here, the subscript $\geq 0$ denotes that this value is capped to be no smaller than zero), and

$$C_i[x](\text{EX}) = C_i(\text{HI})/p - C_i[x](\text{LO}) \tag{9}$$

where $p$ being the number of minor cycles that equal the period of the task and $x$ goes from 1 to $p$.

To illustrate this, a task with $C_i(\text{LO}) = 8$ and $C_i(\text{HI}) = 12$ split over 4 cycles would have pairs of values for $C_i[x](\text{LO})$ and $C_i[x](\text{EX})$ of: (3, 0), (3, 0), (2, 1) and (0, 3).

As all minor cycles are the same following step one, we initially focus on the first cycle. The HI-criticality load from single cycle tasks is added to the extra load from the set of $C_i[1](\text{LO})$ and $C_i[1](\text{EX})$ values. The analysis of the job-based scheme is applied to give values of $S[1]^{\min}$, $\Delta[1]^{\text{HI}}$ and $\Delta[1]^{\text{LO}}$. If the size of the minor cycle is $D$, $S[1]^{\min} + \Delta[1]^{\text{HI}} \leq D$ and $S[1]^{\min} + \Delta[1]^{\text{LO}} \leq D$ then the first cycle is schedulable and the scheme moves on to the second cycle.

However if the first cycle is not schedulable then the next step is to fill the makespan (if there are 'gaps') by moving work from $C(\text{EX})$ to $C(\text{LO})$. Again this follows the job-based approach. If this is not sufficient then work needs to be moved from some task's (or tasks') $C_i[1](\text{LO})$ to $C_i[2](\text{LO})$ so as to reduce $S[1]^{\min}$.

Once $C_i[1](\text{LO})$ and $C_i[2](\text{LO})$ have changed then the relevant $C_i[1](\text{EX})$ and $C_i[2](\text{EX})$ values are recomputed.

To make the first cycle schedulable any task that is active in the following cycle may be chosen as the one to have its work moved from the first to the second cycle. The task or tasks to choose are those that will not have their criticality behaviour in the first cycle changed. This constraint is best illustrated by an example. If a task with $C(\text{LO}) = 5$ and $C(\text{HI}) = 10$ is split into two then all of the following schemes are valid for the two computation times in the two cycles: (5,0) and (0,5), or (5,1) and (0,4) etc. until (5,5) and (0,0). But if this task moves just a single unit of its LO-criticality execution requirement into the second cycle then the only valid scheme is (4,0) and (1,5).

Once the first cycle is made schedulable the process is repeated on the second and subsequent cycles. If in any cycle there is no available work to be moved forward (to another cycle), then the technique of moving work within a cycle from after the switching point to before may be attempted. If none of these schemes work then the task set is not schedulable.

Intuitively, work in being moved forward until $C(\text{LO})$ is satisfied. Then a task's work can be done as either $C(\text{LO})$ or $C(\text{EX})$ which gives more flexibility. So if with the example used earlier, with $C_i[x](\text{LO})$ and $C_i[x](\text{EX})$ values of: (3, 0), (3, 0), (2, 1) and (0, 3), only two ticks could be accommodated in any $S[x]^{\min}$ then work would be pushed through until the following is obtained: (2, 0), (2, 0), (2, 0) and (2, 4).

If the HI-criticality tasks can be allocated then the next step is to allocate the LO-criticality tasks. From the HI-criticality stage $k$ switching times have been computed $S[1] \ldots S[k]$. The available space is therefore $D - S[1] + D - S[2] + \cdots + D - S[k]$. LO-criticality tasks are spread evenly across the cycles. Those that execute every cycle must go into every cycle, those that execute every two need to be spread across the first two, then third and fourth etc. This is continued until, again, the allocation is successful or the task set is deemed unschedulable.

### 4.1 An example

We illustrate our technique for the following task set:

| | $\chi_i$ | $T$ | $C_i(\text{LO})$ | $C(\text{HI})$ |
|---|---|---|---|---|
| $\tau_1$ | HI | 10 | 2 | 3 |
| $\tau_2$ | HI | 10 | 3 | 4 |
| $\tau_3$ | HI | 10 | 2 | 3 |
| $\tau_4$ | HI | 10 | 1 | 2 |
| $\tau_5$ | LO | 10 | 2 | 2 |
| $\tau_6$ | LO | 10 | 3 | 3 |
| $\tau_7$ | LO | 10 | 1 | 1 |
| $\tau_8$ | HI | 20 | 4 | 6 |
| $\tau_9$ | HI | 20 | 6 | 8 |
| $\tau_{10}$ | LO | 20 | 2 | 2 |

Here there are two criticality levels (HI and LO) and the minor cycle time is 10ms. Tasks have periods of either 10 or 20, so only two minor cycles are needed

in the system's major cycle. The hardware platform has two cores, so each of the two minor cycles contains two frames.

The two HI-criticality tasks with period of 20 must be split. So $\tau_8[1]$ has a computation time $C_8[1](\text{LO}) = 6/2 = 3$; and $\tau_9[1]$ has computation time $C_9[1](\text{LO}) = 8/2 = 4$. Adding 3 and 4 to the computation times of tasks $\tau_1$ to $\tau_4$ gives a makespan for $S[1]^{\min}$ of (2+3+2+1+3+4)/2 = 7.5.

The value of $\Delta[1]^{HI}$ is 2 which is acceptable, but the makespan for the LO-criticality jobs ($\Delta[1]^{\text{LO}}$) is 3. So $S[1]^{\max}$ is 7 and $S[1]^{\min} > S[1]^{\max}$ which breaks the invariant for schedulability. We need to move work out of the first cycle so that $S[1]^{\min}$ has a value of 7.

Choosing $\tau_9$ we reduce its computation time in the first cycle to 3. This means that $S[1]^{\min}$ now has a makespan of (2+3+2+1+3+3)/2 = 7, which is acceptable.

In the second cycle $\tau_8$ needs 1 in the LO-criticality mode and 2 more in the HI-criticality mode (ie. $C_8[2](\text{LO}) = 1$ and $C_8[2](\text{EX}) = 2$). Task $\tau_9$ now needs 3 in LO-criticality mode and 2 more in HI-criticality mode. So $S[2]^{\min}$ now has a makespan of (2+3+2+1+1+3)/2 = 6, and $\Delta[2]^{\text{HI}}$ in the second cycle is (1+1+1+1+2+2)/2 = 4. So $S[2]^{\min} + \Delta[2]^{\text{HI}}$ is 10, which is the upper bound.

Finally we add $\tau_{10}$. There is no room in the first cycle, but it can be added to the second cycle. The value of $\Delta[2]^{\text{LO}}$ for the second cycle is now 4, from (2+3+1+2)/2, which is just acceptable as now $S[2]^{\min} + \Delta[2]^{\text{LO}}$ is again 10.

The analysis shows that the full task set is schedulable over two frames and two cycles with switching times of 7 in the first cycle and 6 in the second. In total 40 time units are required (2 x 2 x 10). If one ignores the benefits of mixed criticality scheduling then the total requirement using $C(\text{HI})$ values is 52. This equates to the use of three frames (that is three cores) which is one core more than is required with criticality-aware scheduling.

## 5 Conclusions and Further Work

Single processor safety-critical systems are often constrained so that they can be implemented as a series of frames in a repeating cyclic executive. In this paper we have extended this approach to incorporate multi-core platforms and mixed criticality applications. We allow a minimum number of tasks to be split across frames and cycles, and propose a practical means of constructing the necessary cyclic schedule.

Under further work we will extend the use of Linear Programming from job-based to task-based scheduling. We will also look to demonstrate how the proposed model can be implemented in Ada. The Ada programming language provides support for various forms of scheduling on single and multiprocessor platforms. This includes direct support for a barrier synchronisation protocol, and controlled task migration. These features, together with execution-time monitoring and timing events, should enable the full model to be represented in Ada. Once an appropriate multicore platform with full Annex D Ada support is available a demonstrator will be implemented.

## References


1. S. Baruah and A. Burns. Fixed-priority scheduling of dual-criticality systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 173–181, New York, NY, USA, 2013. ACM.
2. S. Baruah and A. Burns. Achieving temporal isolation in multiprocessor mixed-criticality systems. In *Proceedings of the 2nd International Workshop on Mixed Criticality Systems (WMC)*, Rome (Italy), December 2014.
3. S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.
4. I. Bate and A. Burns. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Systems*, 25(1):5–37, 2003.
5. A. Burns and S. Baruah. Semi-partitioned cyclic executives for mixed criticality systems. In *Proceedings of the International Workshop on Mixed Criticality Systems (WMC)*, December 2015.
6. A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *Proceedings of 27th ECRTS*, pages 3–12, 2015.
7. T. Fleming, S. Baruah, and A. Burns. Improving the schedulability of mixed criticality cyclic executives via limited task splitting. In *Proc. 24th International Conference on Real-Time Networks and Systems*, pages 277–286, 2016.
8. T. Fleming and A. Burns. Extending mixed criticality scheduling. In *Proceedings of the International Workshop on Mixed Criticality Systems (WMC)*, December 2013.
9. 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)*, pages 17:1–17:15, Montreal, Oct 2013.
10. R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.
11. L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press, Dec. 1986.
12. D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed critical scheduler. In *Proceedings of the International Workshop on Mixed Criticality Systems (WMC)*, pages 67–72, December 20143,.
13. D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.
14. D. Tamas-Selicean and P. Pop. Task mapping and partition allocation for mixed-criticality real-time systems. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pages 282–283, Dec 2011.
15. S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.