Improving the Schedulability of Mixed Criticality Cyclic Executives via Limited Task Splitting

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

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

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

ABSTRACT

Mixed Criticality workloads present a challenging paradigm which requires equal consideration of functional separation and efficient platform usage. As more powerful platforms become available the consolidation of previously federated functionality becomes highly desirable. Such platforms are becoming increasingly multi-core in nature bringing challenges in addition to those of isolation and utilisation. Cyclic Executives (CE) are used extensively in industry to schedule highly critical functionality in a manner which aids certification. The CE paradigm may be applied to the mixed criticality case making use of a number of features to ensure the sufficient separation of different levels of criticality. While previous work has considered the separation of criticality levels, this work focuses on providing high system utilisation. One of the significant challenges of such an implementation is the allocation of work (tasks) to minor cycles and cores. This work considers such an allocation problem and presents a means of testing schedulability using Linear Programming (LP) tools. Toward the aim of high system utilisation we consider how tasks of different criticality levels might be split, in some limited way, in order to increase the overall schedulability. We show that even minimal task splitting can drastically release slack previously unusable due to isolation requirements, which in turn provides a significant increase in schedulability.

1. INTRODUCTION

Mixed Criticality (MC) systems are one of the key challenges in current real-time study. A mixed criticality system contains work of two or more differing levels of criticality. Such systems are becoming more common as functionality increases in complexity and more powerful computing platforms become available. This leads to the desire to consolidate previously federated functionality onto a unified platform in order to reduce factors such as cost, weight and power. Vestal's seminal work in 2007 [22] started much of the mixed criticality study to date, a comprehensive review of which can be found in [6].

One of the most widely used scheduling paradigms in industry is the Cyclic Executive (CE) [1]. CEs are favoured, particularly in safety critical domains due to their simplicity and high level of predictability. This makes them an excellent choice for systems that are subject to stringent safety constraints and requirements. A standard CE is made up of a static schedule of repeating code, more specifically the major cycle of duration T^M repeats in a cyclic manner and is composed of a number of minor cycles (each of duration T^F), each minor cycle contains an allocation of work. These minor cycles execute sequentially until the end of the major cycle, at this point execution of the cycle restarts. The lengths of the minor and major cycles are determined during the design of the system – typically the major cycle is a multiple of the minor cycle (e.g. $T^F = 25$ms & $T^M = 100$ms). These decisions are made during system design by taking into account the attributes and requirements of the tasks relying on the structure of the CE, such as

- Tasks must have periods that are multiples of the minor cycle.
- Tasks must have periods that are no greater than the major cycle.
- Tasks must have deadlines greater than or equal to the minor cycle.

It is clear from these constraints that the construction of the system and the cyclic executive itself is a tightly coupled process. However, the trade-off is a highly predictable system which is advantageous when certification is required.

Complicating the landscape further, there is no denying the increasing prevalence of multi-core architectures. While systems executing at the highest levels of criticality are still dominated by single processor architectures, this is becoming increasingly unsustainable as the demand for additional functionality and thus Mixed Criticality options increases. Leveraging the power of multi-core architectures will be required in order to keep pace with new functionality. While this work does not seek to tackle many of the open problems with multi-core systems, it does seek to combine the well established CE with challenges of multi-core and mixed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

RTNS '16, October 19-21, 2016, Brest, France

© 2016 ACM. ISBN 978-1-4503-4787-7/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2997465.2997492

criticality workloads.

One of the fundamental requirements of a mixed criticality system is the maintenance of isolation between differing criticality levels. In the multi-core context this isolation must consider not only the allocation of tasks, but also the inter-core communication and resource access. One approach is to make use of a barrier protocol [11]. This mechanism, which requires hardware or minimal OS support, provides a means of ensuring that at any given time tasks of only a single criticality level are executing across all cores. It is clear that this creates an allocation problem, where tasks must be assigned to the appropriate number of minor cycles, in addition to a processing core, all the while ensuring that tasks of differing criticality levels remain separated. Further importance is placed upon the discovery of an effective allocation due to the potential increase in overheads involved with the introduction of a barrier protocol.

Such an allocation problem for a cyclic executive, even without the mixed criticality and multi-core aspects is known to be NP hard in its complexity. Linear Programming (LP) tools can be used to contend with this allocation problem. The work reported in this paper makes use of the LP tool Gurobi [13] to model the system requirements and produce a feasibility/schedulability test. The use of such a tool is advantageous as it will always generate an optimal result if one exists. While the potentially high computational cost is usually a weighty trade-off when using LP tools, in our work we only seek feasibility or very simplistic optimisation. As such, the timing performance is, in general, very good.

One of the key criticisms of the CE scheduler is that it inefficiently utilises system resources. In a MC multi-core context this is even more relevant as we utilise a barrier protocol which requires all cores to be executing work of the same criticality level. If no work is available on a core, it may have to idle until the barrier signals the release of the next level of criticality. We address this problem by allowing limited task splitting. We model this using Mixed Linear Programming (MLP) which allows the use of both integer and continuous variables. We consider task splitting at the scheduling level, i.e. tasks are suspended and resumed, code is not physically separated during design.

In this work we show that by allowing tasks to split we can greatly increase the schedulability of a mixed criticality cyclic executive multi-core platform. We will illustrate how to split lower and higher criticality level tasks, and how to deal with those criticality levels with multiple execution time predictions. We make use of LP tools to provide an efficient feasibility test and ensure that task splitting only occurs when necessary.

The remainder of this work is structured as follows: Section 2 discusses some related work, Section 3 details the system model, Section 4 describes the basic mixed criticality cyclic executive approach, Section 5 discusses the splitting of lower criticality tasks, Section 6 provides insight into the function of the LP solver, Section 7 discusses the splitting of higher criticality tasks, Section 8 presents an experimental evaluation and Section 9 presents some conclusions.

2. RELATED WORK

Baruah and Burns [2] began work using the barrier approach suggested in [12]. This approach, which requires hardware or minimal OS support, is used as a means to separate the execution of different criticality levels. The barrier waits for each core to call it indicating that the work at the current criticality level has completed, once all cores have completed, work of the next criticality level may commence execution.

Burns et al. [7] consider the use of the barrier protocol in the context of a cyclic executive schedule. Relevant to this work they consider the allocation of tasks via heuristics. Their allocation is simplified by assuming the system only has a single minor cycle ($T^F = T^M$), as such it is just an allocation of tasks to cores, not minor cycles. They consider the performance of First Fit (FF), Worst Fit (FF) and First Fit with Branch and Bound (FFBB)[1]. The investigation illustrated the cost of using the barrier approach, however, they concluded that the advantage of strong separation outweighed the reduced schedulability.

Fleming and Burns [10] continued on from the work in [7] by considering a more complete CE with multiple minor cycles and comparing the heuristic allocation approaches with an optimal LP model. The addition of multiple minor cycles created a more complex allocation problem than the single cycle model. They showed that, while the heuristics provided very good approximations of the ILP result in the single cycle case, in the multi-cycle case the ILP schedulability test provided a large improvement. They also investigated the relative runtime cost of each approach showing that ILP can be used efficiently as a feasiblity test.

Sigrist et al. [15] provide a comparison between static and dynamic allocation techniques. They investigate the scalability of static and dynamic globally scheduled approaches over many cores and show that static approaches remain competitive with regard to overheads.

Huang et al. [14] present an isolation scheduling policy which makes use of the prior work in [12] to evaluate a number of scheduling schemes.

Tamas Selicean and Pop [18, 17] also considered the issue of static task mapping. The work in [16, 19, 20, 3] investigates Time Triggered approaches which manage mixed criticality workloads by switching to pre-computed tables in the event of a criticality change. However, the work above only focuses on single processor platforms.

3. THE SYSTEM MODEL

The system model revolves around the two fundamental components of a cyclic executive, the minor cycle (T^F) and the major cycle (T^M). The major cycle is made up of a number of minor cycles, these minor cycles tend to be of equal size. The major cycle repeats its execution in a cyclic manner. Figure 1 shows the basic structure of a cyclic executive with 4 minor cycles:

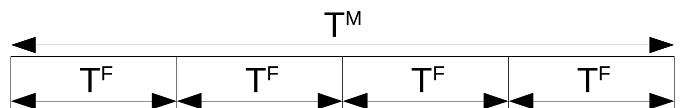

Figure 1: Cyclic Executive Structure.

A task τ_i, in a dual criticality system, is specified as $\tau_i = \{L_i, C_i(LO), C_i(HI), T_i\}$, where L_i is the criticality level (in this work we focus on dual criticality systems using the levels

[1]An initial pass of first fit to identify the point at which the criticality change occurs, followed by a branch and bound search in an attempt to reduce this point.

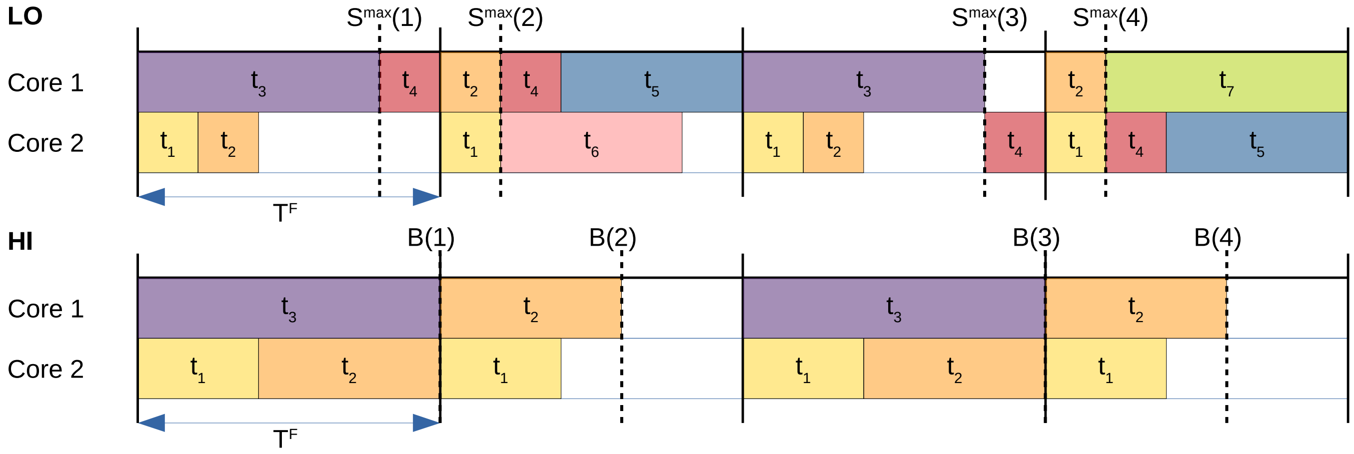

Figure 2: The upper and lower figures depict LO-criticality and HI-criticality execution traces for the example task system of Table 1.

LO and HI; where $LO < HI$), $C_i(LO)$ is the WCET for the LO criticality mode, $C_i(HI)$ is the WCET for the HI criticality mode ($C_i(LO) < C_i(HI)$), T_i is the period.

Tasks are allocated to a single, or a number of, minor cycles, based upon their period (which must be a multiple of the minor cycle). Each task must complete its entire WCET within the minor cycle it is allocated[2] (instances of the same task may be allocated to different cores in subsequent minor cycles). In the non-mixed criticality case, task allocation is simple and schedulability may be determined by simply summing the execution times (C_i) of the tasks allocated to a minor cycle. In the mixed criticality case a barrier protocol [11] is used to completely separate the execution of tasks with differing criticality levels. Within each minor cycle, HI criticality work is allocated first, this is done across all available CPU cores. The point at which all HI work has completed (across all cores) is denoted by $S^{\max}$. The value of $S^{\max}$ is found by allocating HI criticality tasks using their LO WCET. The HI WCETs must be used to check that all HI work may complete by the end of the minor cycle in the event of a criticality change. A criticality mode change occurs when HI criticality work executes to point S^{max} without signalling completion, HI criticality work is allowed to complete its execution in the HI criticality mode, while LO criticality task are suspended. The criticality level returns to *Normal* or LO at the start of the next major cycle (T^M). Once this has been calculated for each minor cycle the LO criticality work is allocated in the same way. Schedulability is established by construction: since the LO execution times (the $C_i(LO)$'s) and the HI execution times (the $C_i(HI)$'s) were used in the allocation, if it is possible to create the schedule, then the instance is schedulable.

More formally, let $S(i, j)$ denote the latest instant at which a core i signals completion of HI work in minor cycle j. Let $S^{\max}(j)$ denote the instant at which LO work may commence across the entire system. Schedulability of a dual criticality, cyclic executive can be calculated as follows (where $HI(i, j)$ and $LO(i, j)$ denote respectively the sets of HI-criticality and LO-criticality tasks scheduled on core i, minor cycle j):

[2]When tasks are not split.

1. Check the schedulability of HI criticality tasks:
$$\forall i \text{ and } j, \sum_{k \in HI(i,j)} C_k(HI) \leq T^F.$$

2. The value of $S(i, j)$ can be calculated for each core:
$$S(i, j) = \sum_{k \in HI(i,j)} C_k(LO)$$

3. The value of $S^{\max}(j)$ used across all cores is:
$$S^{\max}(j) = \max_i(S(i, j))$$

4. LO criticality jobs must fit within the time between $S^{\max}(j)$ and the end of the minor cycle:
$$\forall i \text{ and } j, \sum_{k \in LO(i,j)} C_k(LO) \leq T^F - S^{\max}(j)$$

4. THE MIXED CRITICALITY CYCLIC EXECUTIVE AND PRIOR WORK

The prior work of Fleming and Burns [10] considered how to provide a task allocation and schedulability test for the standard mixed criticality cyclic executive. This section will describe the runtime, illustrate with an example and provide an insight into how Integer Linear Programming was used to find task allocations.

The runtime of the mixed criticality cyclic executive is as follows: execution starts in the first minor cycle, HI criticality work is executed across all cores until point $S_1^{\max}$, if all work has completed by this point, LO criticality work is executed. If the HI criticality work has not completed, it is allowed to execute up to its HI WCET. While in this form the criticality change is strict, in practice it is unlikely that HI work will require execution much beyond $S^{\max}$ if a criticality change occurs. Once this work has completed, LO criticality work is still released but is not guaranteed to complete. This degree of dynamic behaviour is handled by the barrier protocol. As such any invocation of the barrier protocol away from time S^{max} will be termed B. Upon completion of the first minor cycle, execution commences on the second minor cycle which is scheduled in a manner similar

to the first, and so on until the last minor cycle, after which execution cycles back to the first minor cycle.

To further explain the runtime of the standard mixed criticality cyclic executive and the contributions of this work we make use of the example task set shown in Table 1.

This task set is to be allocated to a platform with two cores and a schedule with four minor cycles.

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	15	25	HI
τ_3	20	25	50	HI
τ_4	5	-	25	LO
τ_5	15	-	50	LO
τ_6	15	-	100	LO
τ_7	20	-	100	LO

Table 1: The Mixed Criticality CE example task set.

Figure 2 shows possible allocations for the task set in Table 1 in both the LO and HI criticality modes, it also indicates the points of $S^{\max}(j)$ in the trace with no criticality change (LO) and the points at which the barrier would be invoked away from S^{max}, point $B(j)$, for the trace showing the criticality change (HI). The Figure shows the extreme case in the HI mode in that each HI task τ_i executes for a duration $C_i(HI)$; however, it may not be the case that all HI criticality tasks execute to their full $C_i(HI)$ values, thus the invocation of the barrier ($B(j)$) may occur earlier in practice. Additionally, this Figure does not show LO work executing after $B(j)$ in the HI case, in reality it would do so, although it may be observed that in this extreme case that work would struggle to complete. In fact, in this example, the only LO task that would execute to completion would be τ_4, and that, too, only during minor cycles 2 and 4.

The work in [10] made use of Integer Linear Programming (ILP) using the Gurobi solver [13]. ILP was used to model task sets and determine schedulability by searching for a feasible allocation. In short, this was achieved by modelling each possible task location (on each core and during each minor cycle) as a binary variable: if that variable is set to 1 then the task is scheduled in that location, if it is set to 0 then it is not. The major part of the model concerns the constraints placed upon these variables. The first set of constraints ensure that a task may only be allocated the correct number of times, within the correct number of minor cycles. For example, the .lp format (a user-readable format produced by Gurobi) produces the following constraints:

- For a generic task τ_i where $T_i = 2 \times T^F$ (the period is equal to twice the minor cycle length):

$$T_i_11 + T_i_21 + T_i_12 + T_i_22 = 1$$
$$T_i_13 + T_i_23 + T_i_14 + T_i_24 = 1$$

- For task τ_3 in Table 1 the constraints are:

```
T3_11 + T3_21 + T3_12 + T3_22 = 1
T3_13 + T3_23 + T3_14 + T3_24 = 1
```

Here each variable is represented in the format:

- $T[taskNumber]_[core][minorCycle]$

Since these are all specified as being binary variables (i.e., they may only take on the values zero or one), these constraints specify that τ_3 be scheduled once during cycles 1 and 2, and once during cycles 3 and 4.

The second major set of constraints looks to ensure the schedulability of any resulting allocation. The first section seeks to ensure schedulability of HI criticality tasks executing up to their $C(HI)$ execution times. All possible allocations on all combinations of cores and cycles are checked. For example, the schedulability of the HI criticality mode during, minor cycle 1, core 1 is specified as follows:

- For the generic HI criticality tasks τ_i and τ_l :

$$C_i(HI) \times T_i_11 + C_l(HI) \times T_l_11 \leq T^F$$

- For tasks τ_1,τ_2 and τ_3 from Table 1:

```
10 T1_11 +15 T2_11 + 25 T3_11  <= 25
```

Following this, the model seeks to determine the value of $S_j^{\max}$ for each minor cycle j. This is done in a similar way, this time using the $C_i(LO)$ WCET values and including an X variable, the same X variable $X[minorCyle]$ is included across all cores of a single minor cycle. These X variables represent the remaining capacity available on all cores to schedule LO criticality work. In this way the synchronised switching provided by the barrier protocol is modelled. The examples below shows X1 being included in the statement for minor cycle 1, core 1:

- For generic HI criticality tasks τ_i and τ_l:

$$C_i(LO) \times T_i_11 + C_l(LO) \times T_l_11 + X_1 \leq T^F$$
$$C_i(LO) \times T_i_21 + C_l(LO) \times T_l_21 + X_1 \leq T^F$$

- For tasks τ_1,τ_2 and τ_3 from Table 1:

```
5 T1_11 + 5 T2_11 + 20 T3_11 + X1 <= 25
5 T1_21 + 5 T2_21 + 20 T3_21 + X1 <= 25
```

The X variables are used to check the schedulability of the LO criticality tasks. The model extract below shows the $X1$ variable being used to check the schedulability of LO criticality work during minor cycle 1:

- For generic LO criticality tasks τ_z and τ_x:

$$C_z(LO) \times T_z_11 + C_x(LO) \times T_x_11 - X_1 \leq 0$$
$$C_z(LO) \times T_z_21 + C_x(LO) \times T_x_21 - X_1 \leq 0$$

- For tasks τ_4,τ_5 and τ_6 from Table 1:

```
5 T4_11 + 15 T5_11 + 15 T6_11 +
20 T7_11 − X1 <= 0
5 T4_21 + 15 T5_21 + 15 T6_21 +
20 T7_21 − X1 <= 0
```

In our examples we will make use of a standard set of parameters which you can see used in the example task set in Table 1. The minor cycle is of length 25 (i.e., $T^F = 25$) and the major cycle is of length 100 (i.e., $T^M = 100$), yielding a cyclic executive with 4 minor cycles. Tasks to be scheduled may have periods of either 25, 50 or 100 and as such must be placed in the correct number of minor cycles. Tasks where $T = 25$ must be scheduled once per minor cycle, tasks where $T = 50$ must be scheduled once in cycles once and two, and once in cycles three and four, finally tasks where $T = 100$ need only be scheduled once per T^M.

It is clear from the example schedules in Figure 2 that task allocation is none trivial and is complicated further by the introduction of the barrier protocol. In the remainder of this

paper we work on allowing task splitting to try and alleviate this issue and leverage more of the available platform. We begin by considering the simpler case of how LO criticality tasks may be split in Sections 5 & 6, and continue to discuss how HI criticality work may be split in Section 7.

5. SPLITTING LO CRITICALITY TASKS

A first step when considering how to allow task splitting is to begin with the simpler case of splitting LO criticality tasks. LO tasks have only a single WCET which makes splitting them straightforward. As mentioned above in this work we are only considering splitting across minor cycles, not across cores. Task splitting is handled via pre-emption, not by any form of manual code splitting — while in practice this introduces overheads, in this work we assume no context switching overheads[3]. Splitting in this manner allows for increased flexibility in how tasks may be split.

Based on the system model where $T^F = 25$ and $T^M = 100$ it is clear that only tasks with periods of 50 or 100 can be split across minor cycles. The splitting behaviour is best illustrated by looking again at the example task set in Table 1. Consider the situation where the LO WCET of τ_7 is now equal to 35, $C_7(LO) = 35$. Clearly τ_7 is not schedulable over a single minor cycle (since $C_7(LO) > T^F$), and therefore must be split across multiple cycles, this is illustrated in Figure 3. In the example τ_7 is able to leverage additional LO criticality capacity by splitting its execution across each of the 4 minor cycles in the system. In this way the splitting of LO criticality tasks allows for more efficient use and ultimately a greater number of schedulable task sets.

We extend the work from [10] by utilising Mixed Linear Programming (MLP) to model a set of tasks as before, however, this time selected low criticality tasks may be split. Splitting is modelled by changing the variable type that indicates the location of a task, from binary to continuous. In this model several variables indicating task locations may contain a portion of a task, for example if a task is split equally across two minor cycles, the variable representing each location will contain the value 0.5.

Along with the constraints required in the base ILP model, some additional constraints are now needed to account for and control the splitting behaviour. The first of these constraints is required to ensure that instances of a split task will execute upon the same core over all of the minor cycles:

- The constrains for generic LO criticality task τ_z are as follows:

$$T_z_11 + T_z_12 + T_z_13 + T_z_14 - Y_1 = 0$$
$$T_z_21 + T_z_22 + T_z_23 + T_z_24 - Y_2 = 0$$
$$Y_1 + Y_2 = 1$$

- The constraints required for τ_7 are as follows:

```
T7_11 + T7_12 + T7_13 + T7_14 − Y1 = 0
T7_21 + T7_22 + T7_23 + T7_24 − Y2 = 0
Y1 + Y2 = 1
```

The values Y1 and Y2 are both binary variables, the sum of both is required to equal 1. As the original statements must equal 0, all split components of a task must be assigned only one of the two cores in order to meet all constraints.

[3]The inclusion of context switching overheads is left for future work.

The second additional set of constraints works to control the continuous variables to ensure the resulting split execution times do not violate the notion of discrete time. The problem is as follows, suppose one of the split instances of τ_7 has the variable 0.38 associated with it, in order to work out the resulting execution time in that particular minor cycle we calculate $20 \times 0.38 = 7.6$. However, this splits a unit of time, thus this split is not permissible. To solve this the following statements are used:

- For τ_z:

$$C_z(LO) \times T_z_11 + C_1 = T^M$$
$$C_z(LO) \times T_z_21 + C_2 = T^M$$
...
$$C_z(LO) \times T_z_24 + C_8 = T^M$$

- For τ_7:

```
20 T7_11 + C1 = 100
20 T7_21 + C2 = 100
...
20 T7_24 + C8 = 100
```

Each variable, which represents a possible location of a split task, is included in a constraint that seeks the sum of the resulting execution time plus an integer variable. The integer variable has a lower bound of 0 and an upper bound of the length of the major cycle. The summation is required to equal the major cycle length, in this case 100 (the exact value has no impact assuming it is suitably large), in this way the split task is constrained to prevent violations of *units* of time. This is repeated for all split tasks.

As mentioned above, we currently require no objective function in order to carry schedulability tests, this results in the ability to leverage a property of the tool. Due to the construction of our model, mostly with regard to the constraints, even if some LO criticality tasks are allowed to split, they will only do so if no pure integer solution exists. As splitting is undesirable unless absolutely required this is highly advantageous. More information on this behaviour is covered in Section 6.

Allowing selected LO criticality tasks to split provides a significant increase in schedulability (as shown in the experiments, see section 8) by tightly packing execution within the LO criticality mode.

6. CONTROLLED SPLITTING

As noted above, in our MLP models we moved away from using binary values to represent task locations to continuous variables but constraining them to be ≥ 0 and ≤ 1. This allows the solver to split selected tasks. In this work, tasks are split across minor cycles only, as splitting across cores might cause concurrent execution of the same sequential tasks. To support splitting via cores an additional mechanism would be required to prevent this, such as the method used in [5]. Furthermore, suitable constraints must be in place to ensure that splitting happens over the correct number of minor cycles and to ensure that the resulting fraction of a task does not violate the notion of discrete (integer) time units. More details of the model developed, along with its constraints and bounds are provided in Sections 4,5 and 7. In this section, we consider how we can ensure that tasks are only split when absolutely required, otherwise a purely integer solution

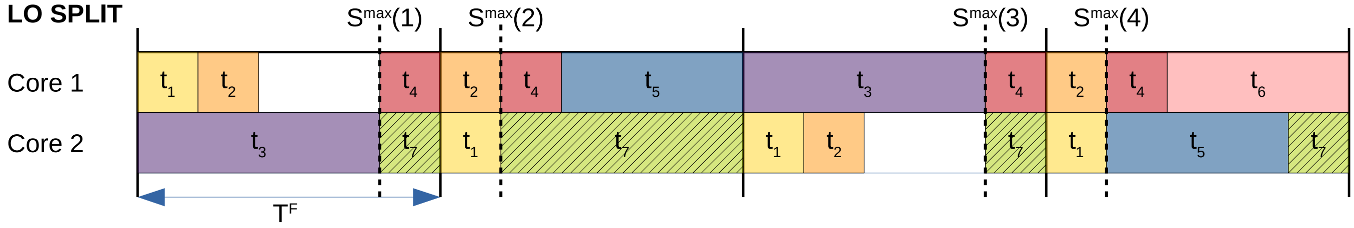

Figure 3: An execution trace in LO criticality behaviour of the example task system in Table 1, with $C_7(LO) \leftarrow 35$. τ_7 is split across the four minor cycles. (Observe that τ_7 is always allocated to the same core.)

will be provided. This is advantageous as splitting is often costly and ought to be performed only when necessary.

The Simplex algorithm of Dantzig [8] is a well established technique for solving linear programs. In brief, the simplex method is an optimisation algorithm that attempts to maximise an objective function. It finds an initial feasible solution and iterates from this point attempting to find the optimal objective function value. Full details of the workings of the algorithm can be found in [21].

In this work we make use of the LP solver (Gurobi [13]) to ascertain the schedulability of a set of tasks. As we are seeking feasibility rather than optimisation, our models are *Phase One solvable* feasibility problems in the sense that the initial feasible solution found during the first phase (before any of the main Simplex iterations) is the only result we seek. To this end we need not specify any objective function. As a consequence we observe that if a model is ILP solvable (i.e. has a feasible integer solution) then the Gurobi solver will produce the same integer result even when the input is specified as being a linear (rather than *integer* linear) program. This is due to it firstly being a feasibility problem (not optimisation, so no objective function) and secondly to the integer values on the bounds and constraints causing the initial feasible solution found by the Gurobi solver to being integer-valued. With the above functionality in mind we can model task splitting and achieve the desirable outcome that tasks are only split if such splitting is required for schedulability. This ensures that no additional overhead is needlessly added. Only if a task set is non-ILP schedulable, will the MILP implementation split tasks as required.

In addition to this behaviour the lack of an objective function offers some insight into the fast performance. As the solver will stop execution once it finds an initial feasible solution, little to no simplex iterations are required. This is the significant factor in the fast performance of the schedulabilty tests.

7. SPLITTING HI CRITICALITY TASKS

Although allowing the splitting of LO criticality tasks goes some way to address schedulability and overall utilisation improvements, clearly the influence that HI criticality tasks have on the barrier synchronisation causes splitting of HI tasks to be desirable in some cases. This section considers how HI criticality tasks may be split, this proves to be significantly more challenging than simply splitting LO criticality tasks.

Two key challenges can be identified:

- How to split tasks with multiple WCET values?
- How and when does the criticality change occur, and what happens to HI tasks?

The problem of splitting mixed criticality tasks was addressed by Fleming and Burns [9] in the context of period transformation. Period transformation is the process of altering the periods of a workload to obtain desirable values. In the mixed criticality case Vestal [22] transformed task periods to achieve a criticality monotonic priority ordering. Although their work considered the splitting of periods, it is applicable to WCET splitting. They note that, a criticality change can only occur, when a task has executed up to its un-split $C(LO)$ value. In this case although the system will be executing in the LO criticality mode, segments of $C_i(HI)/a$ (where a is the factor the task must be split by) must be executed in order to ensure schedulability.

Inspired by our MLP techniques, we consider a task's execution time characterised by two "containers" of size $Ct(LO)$ and $Ct(EX)$, $Ct(LO)$ is a variable where $C(LO) \leq Ct(LO) \leq C(HI)$ and $0 \leq Ct(EX) \leq (C(HI) - C(LO))$. The purpose of these containers is to allow our model solver to increase the amount of execution allocated to the LO container $Ct(LO)$, which in turn will decrease the required execution time in the HI criticality mode (represented by $Ct(EX)$). This provides increased flexibility to the MLP tool and will have a particular impact where the difference between $C(LO)$ and $C(HI)$ is large. In addition if more execution is allocated to $Ct(LO)$ the time at which a criticality change occurs is increased (as it is equal to $Ct(LO)$), in turn this will reduce the likelihood of a criticality change occurring.

The splitting of a HI criticality task is now achieved by splitting both the $Ct(LO)$ and $Ct(EX)$ containers. $Ct(LO)$ may be split among any number of minor cycles (but not cores), the criticality change now occurs at $Ct(LO)$. $Ct(EX)$ may be split also, however split instances of $Ct(EX)$ may only be allocated to the minor cycle at which $Ct(LO)$ completes and any after, as HI criticality execution time only need occur after a criticality change. In this way the solver is provided with the ability to both distribute $Ct(LO)$ and $Ct(EX)$, (within bounds) and split both containers where appropriate.

To summarise, the splitting of a HI criticality task, τ_i, is achieved in the following steps:

- The LO container, $Ct_i(LO)$, is allocated execution time of the task such that $C_i(LO) \leq Ct_i(LO) \leq C_i(HI)$.
- The EX container, $Ct_i(EX)$, is allocated the remaining execution time for the HI criticality mode. $Ct_i(EX) = C_i(HI) - Ct_i(LO)$.

  The task is now represented such that $Ct_i(LO)+Ct_i(EX) = C_i(HI)$.
- $Ct_i(LO)$ may now be split across a number of minor

cycles, each split instance of $Ct_i(LO)$ is represented by $Ct_{ij}(LO)$ where j is the minor cycle.

$$Ct_i(LO) = \sum_{j \in T^M} Ct_{ij}(LO)$$

Each instance of $Ct_{ij}(LO)$ may be of differing lengths, such an allocation is illustrated in Figure 4.

Figure 4: An allocation of $Ct_{ij}(LO)$ for τ_i.

- Following the allocation of $Ct_i(LO)$, $Ct_i(EX)$ must be allocated. Each instance is again described as $Ct_{ij}(EX)$ where j is the minor cycle.

$$Ct_i(EX) = \sum_{j \in T^M} Ct_{ij}(EX)$$

Split instances of $Ct_i(EX)$ may only be allocated the minor cycle, and all those following the final allocation of $Ct_i(LO)$. Figure 5 displays the final allocation.

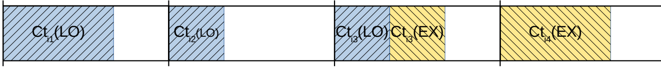

Figure 5: The Final allocation of both $Ct_{ij}(LO)$ and $Ct_{ij}(EX)$ for τ_i.

Consider the example used to illustrate LO criticality task splitting in Figure 3. This example uses the task set in Table 1 to produce a schedule using ILP. The resulting schedule in Figure 2 shows 15 units of LO criticality execution time (in 3 blocks of 5 units of time each) spare and 20 units of spare HI criticality execution. By allowing τ_3 to split, this slack can be moved to the LO side of the barrier, this allows for an additional task, τ_8. Figure 6 presents the resulting schedule. The schedule shows τ_3 splitting across minor cycles 1 and 2 to accommodate τ_8, however τ_3 does not split over minor cycles 3 and 4 as it is not required.

The approach used to model this behaviour is an extension to the idea of considering each possible location a task may take as a variable (binary or continuous). To re-cap, the initial approach, using ILP, consisted of a number of binary variables representing task locations, the variable for τ_i core 1, minor cycle 1 is Ti_11. Constraints are specified to limit the possible allocations based on task frequency and other factors. When splitting LO tasks these variables become continuous with a lower bound of 0 and an upper bound of 1. This allows for splitting as work might then be allocated across multiple variables, for example core 1 cycle 1 Ti_11 and core 1 cycle 2 Ti_12. If the task was split 50/50 across each of these minor cycles, then each variable would contain the value 0.5. Thus, largely the same constraints as used for the ILP still hold for the MLP version as the requirement that a set of variables equals 1, or a whole task, is still valid (but might be split across many variables). The MLP models which allow HI splitting are only supplied with the $C(LO)$ of each task, however, as described above we must allocate execution time into both $Ct(LO)$ and $Ct(EX)$ in order to provision for $C(HI)$. Rather than the direct value of $C(HI)$ being provided, the model is given $C(HI)$ as a product of $C(LO)$ represented as m. The value of m is calculated before the model is created such that $m = C(HI)/C(LO)$. In this way, rather than the overall task being bounded and required to equal 1, as in the case of the ILP and MLP with low splitting, this work is bounded and required to equal m. Each possible location of the task will contain a portion of m, $m \times C(LO)$ is used to work out this value when considering schedulability. To illustrate this the constraints on the rate of high task τ_3 during minor cycles 1 and 2 will change:

- From

```
Ti_11 + Ti_21 + Ti_12 + Ti_22  = 1
```

- To:

```
Ti_11 + Ti_21 + Ti_12 + Ti_22 +
Ti_11EX + Ti_21EX + Ti_12EX + Ti_22EX = m
```

In the case of τ_3 as its $C_3(LO) = 20$ and its $C_3(HI) = 25$, $m = 25/20 = 1.25$.

The new EX variables are included in the previous stages of the model for those HI criticality tasks that are allowed to split. The listing of the initial statement defining where and how frequently a task can be scheduled will now include these variables:

- The statements for τ_i:

$$T_i_11 + T_i_21 + T_i_12 + T_i_22 + T_i_11EX$$
$$+ T_i_21EX + T_i_12EX + T_i_22EX = m$$
$$T_i_13 + T_i_23 + T_i_14 + T_i_24 + T_i_13EX$$
$$+ T_i_23EX + T_i_14EX + T_i_24EX = m$$

- The statements for τ_3:

```
T3_11 + T3_21 + T3_12 + T3_22 + T3_11EX
+ T3_21EX + T3_12EX + T3_22EX = 1.25
T3_13 + T3_23 + T3_14 + T3_24 + T3_13EX
+ T3_23EX + T3_14EX + T3_24EX = 1.25
```

The statments must equal 1.25 to account for the $C(EX)$ execution time, in other words $C_3(HI)/C_3(LO) = 1.25$. It follows that these EX variables must also be included in the WCET calculations, for the HI mode only (as both $T3_11$ and $T3_11EX$ are required to produce the full HI WCET). In this case, τ_i is allowed to split and τ_l is not:

- For τ_i:

$$C_i(LO) \times T_i_11 + C_i(LO) \times T_i_11EX + C_l(HI) \times T_l_11 \leq T^F$$

- For τ_3, where τ_3 is allowed to split and τ_1 & τ_2 are not:

```
10 T1_11 +15 T2_11 + 20 T3_11 +
 20 T3_11EX  <= 25
```

Finally, the constraints that ensure tasks are split across minor cycles but not cores need alteration. They become:

- for τ_i:

$$T_i_11 + T_i_12 + T_i_11EX + T_iz_12EX + 10 \times Y_1 \geq m$$
$$T_i_21 + T_i_22 + T_i_21EX + T_iz_22EX + 10 \times Y_2 \geq m$$
$$T_i_13 + T_i_14 + T_i_13EX + T_iz_14EX + 10 \times Y_3 \geq m$$
$$T_i_23 + T_i_24 + T_i_23EX + T_iz_24EX + 10 \times Y_4 \geq m$$
$$10 \times Y_1 + 10 \times Y_2 = 10$$
$$10 \times Y_3 + 10 \times Y_4 = 10$$
$$10 \times Y_1 + 10 \times Y_4 = 10$$
$$10 \times Y_2 + 10 \times Y_3 = 10$$

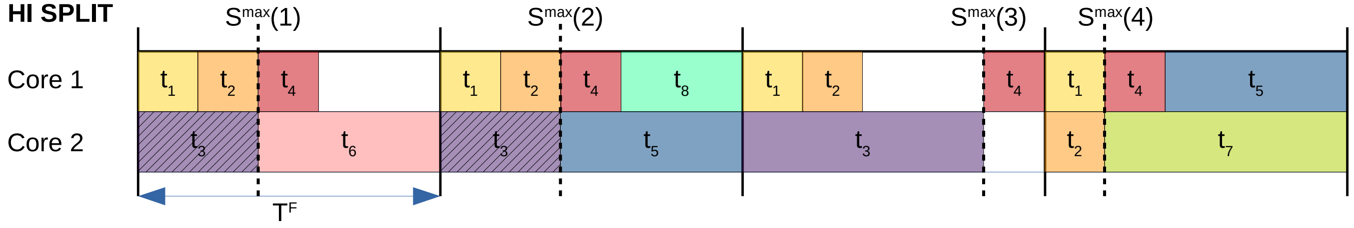

Figure 6: An execution trace of Table 1, where τ_3 is split and τ_8 is added.

- For τ_3:

```
T3_11+ T3_12+ T3_11EX+ T3_12EX+ 10 Y1>= 1.25
...
10 Y1 + 10 Y2 = 10
...
```

These constraints now use the binary Y variables to ensure that the tasks are split only across minor cycles and not CPU cores.

In addition to these changes, new constraints are added. The first seeks to ensure that $Ct(EX)$ may only be allocated in minor cycles where $Ct(LO)$ has just completed, or has already completed. The listing for this constraint is shown:

- for τ_i:

$$T_i_11 + T_i_11EX + T_i_12EX + 10 \times Z_1 \geq m$$
$$T_i_11 + T_i_12 + T_i_12EX + 10 \times Z_2 \geq m$$
$$T_i_13 + T_i_13EX + T_i_14EX + 10 \times Z_3 \geq m$$
$$T_i_13 + T_i_14 + T_i_14EX + 10 \times Z_4 \geq m$$
$$10 \times Z_1 + 10 \times Z_2 + 10 \times Z_3 + 10 \times Z_4 \leq 30$$

- For τ_3:

```
T3_11 + T3_11EX + T3_12EX + 10 Z1 >= 1.25
T3_11 + T3_12 + T3_12EX + 10 Z2 >= 1.25
T3_13 + T3_13EX + T3_14EX + 10 Z3 >= 1.25
T3_13 + T3_14 + T3_14EX + 10 Z4 >= 1.25

10 Z1 + 10 Z2 + 10 Z3 + 10 Z4 <= 30
```

This constraint works by defining each possible combination of $Ct(LO)$ and $Ct(EX)$ allocations while requiring one to be correct using the Z variables. The example shows only those combinations on core 1, the same is repeated for all additional cores with the right hand value of the inequality being $C(HI)/C(LO)$. The requirement of the Z variables to be less than 30, requires that one of the combinations shown be correct.

The second set of constraints is required in order to ensure splitting only occurs when absolutely required. The constraints are shown:

- for τ_i:

$$10 \times N_1 + T_i_11 + T_i_11EX \geq m$$
$$10 \times N_2 + T_i_21 + T_i_21EX \geq m$$
...
$$10 \times N_8 + T_i_24 + T_i_24EX \geq m$$

- For τ_3:

```
10 N1 + T3_11 + T3_11EX >= 1.25
10 N2 + T3_21 + T3_21EX >= 1.25
...
10 N8 + T3_24 + T3_24EX >= 1.25
```

Each of these statements sums an N value with each task location and its EX variable. If a task is completely scheduled within its variable and EX variable, it is not split and runs during the same minor cycle. Thus it will achieve the value of 1.25 when the location and EX variables are summed, otherwise the N value, which is a binary variable will be set to 1 and the overall result will be 10. The solver is provided with a simple minimisation function which seeks to minimise the sum of the N variables.

A note on the MLP schedulability test

As is clear from the structure of the MLP model allowing HI criticality task splitting, the property described in Section 6 (where tasks are split only when an integer solution does not exist due to the initialisation of the simplex method) does not hold in the same sense. While the important property of splitting tasks only when required to do so (when no ILP solution exsists) is maintained, a simplistic minimisation function is required. Although a small loss in performance may be noted, the minimisation approach has additional benefits. While the models which allow LO tasks to split when no integer solution exists, they do not control splitting in any way once it is required (e.g. multiple tasks might be split, where only one needs to be). By using a minimisation function HI tasks are split in such a way that, even when work must be split, the model attempts to minimise the number of split tasks.

8. EVALUATION

In order to better understand the performance gains provided by task splitting we performed an experimental evaluation. Our evaluation consists of a large number of schedulability tests on synthetic task sets over varying utilisation values.

The setup of the experiments is as follows:

- A dual core platform.
- The cyclic executive consists of 4 minor cycles per major cycle, each minor cycle is equal to 2500 with the major being equal to 10,000
- A total of 1000 task sets were generated per 5% increase in utilisation.
- 10 tasks were generated per set.
- Task periods are given as either 2500, 5000 or 10,000.
- Task set utilisation (U) values are generated via UUniFast [4].
- C(HI) execution times are derived by $C(HI) = U \times T$, with $C(LO)$ values equal to $C(HI) \times n$ where n is some scaling factor that maintains $C(HI) \geq C(LO)$.

The initial comparison consisted of comparing the performance of ILP (no splitting), MLP-LO-ALL (only splitting LO tasks) and MLP-HI-ALL (splitting all possible tasks)

to evaluate the gain in schedulability. The results of this comparison can be seen in Figure 7. Figure 7 shows a

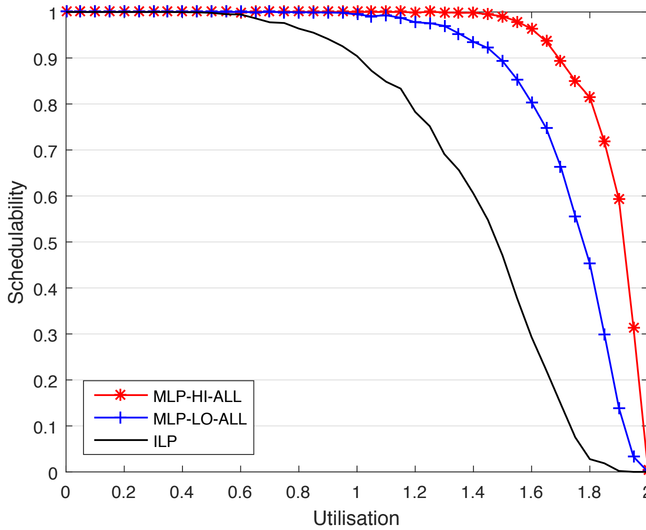

Figure 7: The performance of the ILP and MLP approaches with full splitting.

significant performance increase when splitting is allowed. The largest jump in schedulability is from the standard ILP to the MLP-LO-ALL where only LO tasks are split. This increase is likely due to the ability to split any tasks providing a big advantage allowing the system to split tasks with potentially large computation times in order to create a feasible schedule. The smaller increase from MLP-LO-ALL to MLP-HI-ALL is down to the fact that the advantage of splitting any task is already accounted for with the differences between the ILP and MLP-LO-ALL results. The difference between MLP-LO-ALL and MLP-HI-ALL represents those task sets which specifically require a HI criticality task to be split in order to preserve feasibility.

A similar story is told when increasing the number of tasks per set and CPU cores. Figure 8 shows the performance of ILP, MLP-LO-ALL and MLP-HI-ALL on an 8 core platform where each task-set contains 60 tasks. It is clear that the relative performance of each approach holds when scaling up the parameters.

While the impact of splitting all LO and HI tasks has been illustrated in Figures 7 and 8, it is possible to investigate how splitting just a single LO task and a single LO & HI task can effect the schedulability. This is illustrated in Figure 9

Figure 9 shows the schedulability when only a single LO task (MLP-LO) and a single HI and LO task (MLP-HI) are allowed to split. In this case these tasks are those with the largest WCET at their criticality level ($C(L)$). We compare these with ILP and the results from Figure 7 which are the dashed lines shown. This plot illustrates how by only allowing a single task to split it is possible to significantly increase the overall schedulability. This is particularly notable in the case of splitting a single LO criticality task (MLP-LO vs MLP-LO-ALL) as it is clear that the performance in terms of schedulability is relatively close to the scenario where all tasks are split. In short, this plot illustrates that only a minor amount of splitting is required to gain a significant increase in schedulability.

With regard to the cost of executing such ILP and MLP programs the result from [10] holds. The average execution

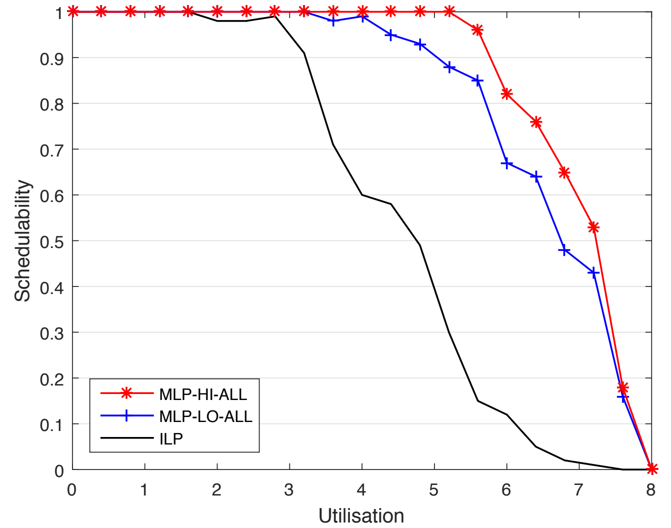

Figure 8: The performance of the ILP and MLP approaches with full splitting on an 8 core platform.

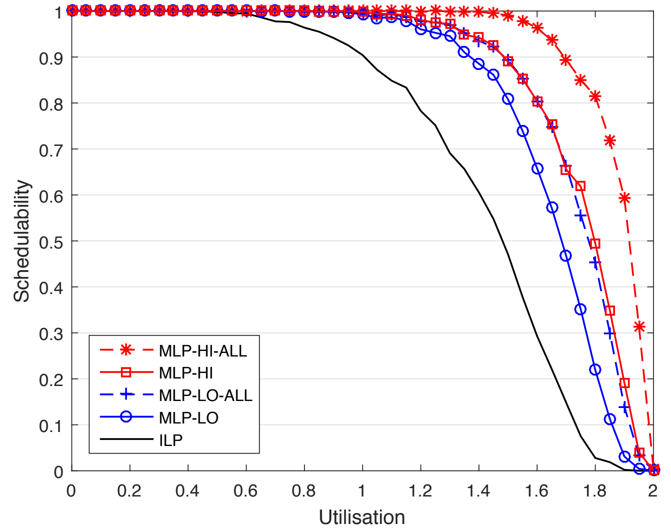

Figure 9: The performance of the ILP and MLP approaches with selected splitting.

time of each approach was recorded for a number of configurations:

- **Cfg1**: 10 Tasks, 2 Cores, 2 Crit levels, 4 minor cycles.
- **Cfg2**: 30 Tasks, 4 Cores, 2 Crit levels, 4 minor cycles.
- **Cfg3**: 60 Tasks, 8 Cores, 2 Crit levels, 4 minor cycles.
- **Cfg4**: 60 Tasks, 8 Cores, 4 Crit levels, 8 minor cycles.

Table 2 presents the results (all values in seconds)[4]. While

	Cfg1	**Cfg2**	**Cfg3**	**Cfg4**
ILP	0.00061	0.00378	0.01053	0.01543
MLP-LO-ALL	0.00078	0.01062	0.14274	0.43472
MLP-HI-ALL	0.01438	0.10225	0.65168	1.0408

Table 2: Timing data from different configurations.

it is clear that the execution time increases as the parameters are scaled up, all average execution times are very reasonable. This supports the conclusion that such LP based feasibility tests may be executed relatively efficiently.

[4]All data gathered from a modern quad core (i7 4790k)

9. CONCLUSION

In this work we have considered the problem of allocating tasks to a mixed criticality cyclic executive system on a multi-core platform. We have shown that while the use of a barrier protocol to separate the execution of different criticality levels does provide isolation, it reduces the overall utilisation by locking potential slack in different criticality modes. We proposed a task splitting approach to deal with this problem, this consisted of two stages:

- LO Criticality: We illustrated how LO criticality tasks may be split in order to re-arrange the available LO slack to schedule additional work. The constraints required of a MLP to express this splitting are detailed and some insight is provided into the performance (and function) of the LP solver.
- HI Criticality: Secondly we present an approach for splitting HI criticality tasks using containers which allow for additional work to be executed in the LO criticality mode if possible. This provides the LP tool with a higher level of flexibility and may provide a decreased likelihood of a criticality change occurring. We illustrate how this can be used to migrate slack from the HI to the LO criticality mode. As before we present the additional constraints required of an MLP model to facilitate HI criticality task splitting.

Finally we present an evaluation which illustrates that with only minimal splitting a large increase in overall schedulability can be achieved. To summarise, both LO and HI criticality tasks may be split in order to efficiently utilise the platform. Linear programming tools provide an effective way of determining such an allocation and provide a schedulability test that splits tasks only when required. Often very minimal splitting can yield large amounts of additional slack, or large increases in schedulability. The work in this paper aims to provide the groundwork for modelling such systems using Linear Programming tools. Significant implementation challenges were tackled, however the solutions are flexible and provide a platform for future work. We believe such work is a necessary step toward more complex allocation and/or optimisation problems in the future.

References

[1] T. Baker and A. Shaw. The cyclic executive model and ada. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 120–129, Dec 1988.

[2] S. Baruah and A. Burns. Achieving temporal isolation in multiprocessor mixed-criticality systems. In *WMC*, page 21, 2014.

[3] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 3 –12, 29 2011-dec. 2 2011.

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

[5] A. Burns and S. Baruah. Semi-partitioned cyclic executives for mixed criticality systems. volume WMC RTSS 2015, pages 7–12, 2015.

[6] A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2016.

[7] A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. ECRTS 2015, 2015.

[8] G. Dantzig. *Linear Programming and Extensions*. Landmarks in Physics and Mathematics. Princeton University Press, 1963.

[9] T. Fleming and A. Burns. Extending mixed criticality scheduling. volume WMC RTSS 2013, pages 7–12, 2013.

[10] T. Fleming and A. Burns. Investigating mixed criticality cyclic executive schedule generation. In *Proceedings of Workshop on Mixed Criticality, IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[11] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15, Sept 2013.

[12] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation & Test in Europe Conference (DATE), Hot-Topic Session on Predictable Multicore Computing*, Dresden, Germany, Mar 2014. IEEE.

[13] I. Gurobi Optimization. Gurobi optimizer 6.0. http://www.gurobi.com/.

[14] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele. An isolation scheduling model for multicores. In *Real-Time Systems Symposium, 2015 IEEE*, pages 141–152, Dec 2015.

[15] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele. Mixed-criticality runtime mechanisms and evaluation on multicores. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 194–206, April 2015.

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

[17] D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 24 –33, 29 2011-dec. 2 2011.

[18] 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.

[19] J. Theis and G. Fohler. Mixed criticality scheduling in time-triggered legacy systems. In *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[20] J. Theis, G. Fohler, and S. Baruah. Schedule table generation for time-triggered mixed criticality systems. In *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[21] R. Vanderbei. *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science. Springer US, 2013.

[22] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239 –243, dec. 2007.