Technical Report

Hard real-time multiprocessor scheduling resilient to core failures

Borislav Nikolic
Konstantinos Bletsas
Stefan M. Petters

CISTER-TR-150405

2015/04/15

Hard real-time multiprocessor scheduling resilient to core failures

Borislav Nikolic, Konstantinos Bletsas, Stefan M. Petters

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: borni@isep.ipp.pt, ksbs@isep.ipp.pt, smp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Most multiprocessor scheduling theory overlooks the possibility of hardware failures that entirely nullify the computation carried out by a task instance, and potentially also make the respective processor henceforth unusable. Yet, such failures may occur, causing the system to fail. Motivated by this reality, we introduce a new concept of hard real-time schedulability guarantees for critical multiprocessor systems and analysis for their derivation. Namely, all deadlines must be met, even in the event of a core failure. A scheduling approach, based on global fixed priorities, and accompanying analysis, for achieving such guarantees are then formulated.

Hard real-time multiprocessor scheduling resilient to core failures

Borislav Nikolić, Konstantinos Bletsas and Stefan M. Petters
CISTER/INESC-TEC, ISEP, IPP, Porto, Portugal
Email: {borni, ksbs, smp}@isep.ipp.pt

***Abstract*—Most multiprocessor scheduling theory overlooks the possibility of hardware failures that entirely nullify the computation carried out by a task instance, and potentially also make the respective processor henceforth unusable. Yet, such failures may occur, causing the system to fail. Motivated by this reality, we introduce a new concept of hard real-time schedulability guarantees for critical multiprocessor systems and analysis for their derivation. Namely, all deadlines must be met, even in the event of a core failure. A scheduling approach, based on global fixed priorities, and accompanying analysis, for achieving such guarantees are then formulated.**

I. Introduction

Hard-real-time multiprocessor scheduling requires offline guarantees of schedulability. Its critical systems subdomain though, requires even stricter schedulability guarantees, in the presence of unlikely events like execution overruns and hardware faults.

For dealing with the former kind of events (execution overruns), and in the context of mixed-criticality systems, a comprehensive toolset of analysis and design techniques is gradually being assembled [11]. However, another kind of rare event endangering a system involves a core failing, temporarily or permanently. This causes whichever task instance (job) was executing on the respective processor to be terminated short of completion as a result, and all the computation performed by it since its arrival to be wasted. Additionally, if this failure is permanent, rather than just temporary, it leaves the system with one usable core less. Even if such a failure is detected immediately, and the terminated task restarted, the schedulability of the system may be compromised because (i) the job restarted from the beginning still has to meet the absolute deadline of the *original* terminated job, which may be too close, and, if the failure is permanent, (ii) the same workload as before has to be scheduled on *fewer* cores.

Motivational example 1: A multiprocessor system with 100 cores has a single task with an execution time of 6 time units and a deadline of 10. It arrives at time $t = 0$ but at $t = 5$ the task is killed due to the core failure. And even if it is restarted immediately, it will miss its deadline at $t = 10$.

Adding processors, in this example, does not help meet deadlines. What *would* help, would be to speculatively launch a redundant copy of the task, in case the original one is killed. But such duplication wastes processing capacity in the general case and even so, does not always bring the desired resilience, as demonstrated with the following example:

Motivational example 2: Consider a multiprocessor system with three cores and two sporadic tasks $\tau_1 = (10, 10, 10)$ and $\tau_2 = (2\epsilon, 10, 10)$, where τ_1 has a higher priority, and the values in brackets denote the execution time, the deadline and the period, respectively. This workload is schedulable (e.g. with global fixed-priorities) even if two copies of each task are released at each arrival. But if both tasks arrive at $t = 0$ and the core where τ_2 runs fails permanently immediately afterwards, a deadline is missed (Figure 1).

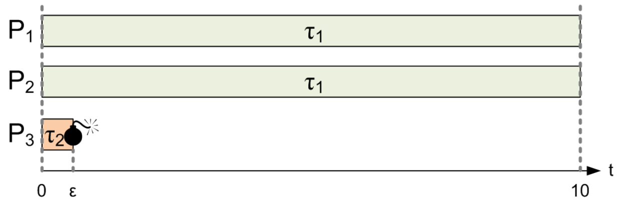


Fig. 1: Accompanying graphic for Motivational example 2.

So ideally, we seek both minimal execution redundancy *and* also offline-provable resilience in case of a core failure. To that end, we propose an approach for scheduling hard-real time tasks resiliently with respect to core failures, based on global fixed priorities. Global scheduling is a natural choice because the single run-queue tends to balance the load on the available processors, smoothing out any transient load due to the failure. It also makes it irrelevant, which is the core that fails, since cores are interchangeable and pooled together – unlike (semi- or full) partitioning. As for fixed-priority scheduling, it is a well-understood widely supported policy.

Core failure semantics: We disregard other kinds of faults (e.g., corruption of main memory) and only consider hardware failures specific to each core (i.e. its ALUs, registers, private caches) and failures in its core-specific software layers (OS, workload). We assume a hardware facility exists for immediate fault detection e.g., the contents of a write-back L1 cache cannot be trusted or the ALU output written to a register is suspect. How this is implemented is beyond the scope of this work, but engineering reality offers examples[1].

Regardless of the failure type (i.e. a transient or a permanent one), a fault indicates that the state of whichever task was executing on the affected processor at detection time, is corrupted. Hence, the task should be aborted, but the original deadline associated with it must still be met – either by a newly launched instance of the task or by a redundant instance already launched speculatively (to cover for a potential failure,

[1]For example, parity bits in L1 caches. Also, in critical systems, pairs of cores are sometimes set to operate in lockstep, with the same inputs [15], effectively being used as a single processor; mismatching core outputs then indicate a fault in one of the two.

and prevent the situation described in Motivational example 1). Conversely, we assume that the state of all other tasks in the system can be trusted, either because they currently execute on non-faulty cores or (if not currently running) because their state resides entirely in main memory, which can be trusted. So all other tasks, except the one currently executing on the processor that failed, are unaffected.

Depending on the criticality of the system, as well as the nature of the fault, it may be reasonable to treat the fault as transient and reuse the core after the task in question has been killed, or alternatively, immediately take the faulty core permanently offline, which leaves the system with one core less. In this work, we consider both possibilities. In any case though, if the system is critical at all, we assume that the operator will seek to take it offline at the first convenient opportunity for doing so in an orderly manner. Therefore, although resilience guarantees under *multiple* core failures are theoretically interesting, in this work we only consider a single failure and believe that this captures most practical scenarios, even if not all. To illustrate, if the mean-time-before-failure (MTBF) for a typical core used in critical systems is hundreds of thousands of hours, the probability of a second core failing within the few minutes needed for safely shutting down the system in a controlled manner, would be too low to be of practical concern in most contexts.

In order for the above semantics and process model to be applicable to a real system, some requirements exist both for the platform and O/S and also in the application design. We already mentioned the need for a facility for detecting failures as soon as they occur. A software facility for starting (and stopping) job copies, as needed, is also required. However, the mere fact that our approach may force a task to, effectively, execute in parallel with itself, may introduce, in the absence of design safeguards against this, synchronisation hazards (i.e., when performing I/O or when accessing variables/manipulating application state) where there were previously none. These and other practical aspects and challenges are discussed again later, in more detail, in Section V.

II. Related work

The multiprocessor scheduling theory has lately received a lot of attention from the real-time community. All state-of-the-art approaches can be broadly classified into several categories.

The *fully-partitioned* approaches are migration-less. Each task is, at design-time, mapped to a particular core, and all its computations have to be performed on that core. Each core has an independent scheduler, which is responsible to schedule the workload located on its respective core in a uni-processor fashion, by implementing one of the scheduling algorithms for uni-processors (e.g. [21]).

In the *semi-partitioned* approaches (e.g. [10], [19]), the tasks are allowed to migrate, however, the migration decisions are derived at design-time. That is, in each iteration, a migrative task must perform a prescribed amount of work on each of the cores where it migrates, and that has to be performed in an already defined order.

The *global* approaches (e.g. [4], [6]) allow unconstrained workload migrations, which means that each task has the possibility to migrate at any time to any core within the platform. All migration decisions are made at runtime, by a single global scheduler.

The *clustered* approaches (e.g. [12]) present a concept which combines the properties of the fully-partitioned and global approaches. Specifically, all cores are divided into disjoint groups, where each group forms one cluster. A cluster is perceived and treated as an independent system, with the global scheduling policy applied on the cluster-level. Moreover, each task is assigned to exactly one cluster, and it has the possibility to freely migrate within its respective cluster.

In recent years, the researchers focused on another type of approaches, which we refer to as the *arbitrary-partitioned* approaches (e.g. LMM [23], APA [18]). In these approaches, each task has a set of cores where it is allowed to migrate to. Note that fully-partitioned and global approaches are only a special case of arbitrary-partitioned approaches, where, in the former case, the core-candidate-set of each task has only one element (core), while, in the latter case, the core-candidate-set contains all elements (cores).

As already mentioned, the focus of this work is on multiprocessors with global fixed priorities. For such a model, Lundberg [22] proposed the response-time analysis. Subsequently, Bertogna and Cirinei [8] proposed an analysis which improves the above result. The improvement is twofold. First, the authors derived a tighter upper-bound on the workload that higher-priority tasks can generate within the analysed time interval. Second, they observed that if an interfering task is ”too large”, not all its workload will necessarily cause interference, as some parts of it might be executed in parallel with the analysed task.

Guan et al. [17] further improve the above result. Their analysis is inspired by the aforementioned method of Bertogna and Cirinei [8], upon which they apply concepts similar to the window analysis framework proposed by Baruah [5]. Specifically, this approach allows to derive even tighter upper-bounds on the workload generated by higher-priority tasks, and consequently derive tighter response-time estimates. This is achieved by proving an upper-bound on the number of interfering tasks that can have *carry-in workload* (defined as workload from jobs released earlier than the start of the time window in consideration) in the worst-case scenario. Recently, Sun et al. [26] proposed an improvement over the analysis of Guan et al. [17], which is applicable to task sets with arbitrary deadlines.

Until now, core failures and their implications on the schedulability analysis have received very little attention from the real-time community. In the uniprocessor domain, one notable work addressing transient core failures is that of Pathan [25]. Specifically, in order to recover from task errors, caused by transient hardware or software faults, the author proposes the release of *backup* jobs. Understandably, permanent processor failures are not covered therein, simply because they are impossible to recover from on a uniprocessor. However, on multiprocessors, this is a possibility. Ghosh et al. [16] proposed a method which allows for multiple core failures, but it is applicable only to task-sets with low-utilised tasks, whereas in our work such a restriction does not exist. Moreover, the works of Pathan [24] and Cirinei et al. [13] consider only transient faults in the multiprocessor setup.

Note that the schedulability analysis with core failures can be also studied from the perspective of *mode changes* (e.g. [20]), whereas the functioning of the system, before and after a core failure, can be perceived as distinctive system modes. Also note that any potentially different workload requirements, before and after a failure, can be analysed with the *mixed-criticality* computational model (e.g. [27]). The state-of-the-art methods from the aforementioned areas indeed efficiently take into account potential workload variations, associated to different system states. However, none of the existing works allows the possibility of variations in available hardware resources, which is an essential requirement for the integration of core failure events in the schedulability analysis. Partly, an exception is the work of Baruah and Guo [7], in the context of mixed-criticality scheduling, which allows degradation in the processor speed. Still, that problem differs from ours in the sense that Baruah and Guo [7] focus on uniprocessors, while in this work we consider multiprocessors. Additionally, in the aforementioned study, the workload is comprised of a finite sequence of independent jobs, whereas we target sporadically recurring tasks.

III. Overview

A. System model

We assume a multiprocessor system with m identical cores and a task-set τ with n sporadic tasks $\{\tau_1, \tau_2, \ldots, \tau_{n-1}, \tau_n\}$. Each task τ_i has a worst-case execution time (WCET) C_i, a relative deadline D_i and a minimum inter-arrival time T_i. We assume constrained deadlines ($D_i \leq T_i$). Tasks are independent, share no resources and are to be scheduled according to a fully preemptive global fixed-priority scheme. Each task has an associated unique priority, assigned at design time, i.e. not given *a priori* as a part of the problem instance.

We use the terms "processor" and "core" interchangeably. We refer to the currently executing job affected by a core failure, and aborted with no salvageable state, as the *wasted* job. A job launched after the detection of a failure, as a replacement instance of the wasted job is termed its *copy job*, with the wasted job being referred to as its corresponding *main job*. For a task that requires redundancy, in order to meet its deadline, even in the case of a core failure, the respective copy jobs may need to be launched speculatively, just in case the corresponding main job is wasted too close to its deadline; then the two may concurrently co-exist in the system. We refer to that co-existence as "overlap" and to tasks for which overlap is possible at run-time as "overlapping". Once a main job by an overlapping task completes, its corresponding copy job is immediately terminated, as it no longer serves any purpose.

Whether a task is overlapping or not, is known offline. It is not part of the input to the design process though; it is simply an arrangement reached at the design stage itself, in order to ensure schedulability.

B. Problem refinement

As mentioned, the system is restricted to global fixed task priority scheduling, with the actual priority assignment originally unspecified. The objective is for all tasks to meet their deadlines, assuming at most one core failure event (whether temporary or permanent). From an abstracted scheduling perspective, the only difference between these two failure types is the number of processors m' that are available after a failure, which is m after a temporary failure, and $m-1$ after a permanent one. Resilience to a core failure is achieved via the use of copy jobs, as described earlier. However, the fact that we only require the system to survive a single core failure event means that, from the perspective of providing such a guarantee, it is no longer beneficial to use copy jobs after "surviving" that single failure. Thenceforth, only main jobs are released, as in conventional scheduling, and no more copy jobs are ever released. For the same reason, all active copy jobs of unaffected tasks, are also dropped, when the core fails.

Under these semantics, the problem amounts to specifying (i) a task priority assignment and (ii) the conditions that trigger the release of copy jobs at run-time, such that the system is provably schedulable even in the event of a core failure.

These decisions are intertwined, since, for example, whether some task needs to be overlapping or not may depend on the set of its higher-priority tasks. To simplify both the decision problem and the dispatching at run-time, we adopt the following arrangement:
– For a non-overlapping task τ_i, a copy job is released at the time of failure of its corresponding main job.
– If τ_i is overlapping, then its copy jobs are released at a fixed (designer-set) offset O_i relative to the arrival of their corresponding main jobs – unless the main job already completes before that time. For tie-breaking, a copy job always has a lower priority than its main job.

Effectively then, in the context of a particular priority assignment, we need to:
(i) Identify the tasks that require speculative copy job execution (i.e. the overlapping tasks) and
(ii) pick appropriate copy job offsets (O_i) for them.

As will be demonstrated with experimental evaluations (Section VI), offsets entail a tradeoff: A small O_i gives the copy job a greater deadline ($D_i - O_i$) within which to complete, hence it facilitates the schedulability of τ_i. However, it increases the interference from speculative redundant computation onto lower-priority tasks.

IV. Schedulability analysis, offset assignment and priority assignmemt

This section describes our proposed approach for scheduling with core-failure-resilience guarantees. First, we introduce an appropriate schedulability analysis, assuming a given priority assignment and a given offset assignment. This analysis is then used to pick the tasks that execute speculatively and their copy job offsets. It can also guide the priority assignment.

Note that the following analysis is general enough to cover either permanent or transitional core fault semantics. The only distinguishing aspect for the two scenarios, from the perspective of the analysis, is the number of available cores post-failure.

A. System mode coverage in analysis

The system operation can be viewed in terms of two modes, S and D. The system starts in Mode S (standard) and immediately switches to Mode D (degraded) when a core failure occurs. Recall that if the failure affected some main job, upon the transition to Mode D, all copy jobs, except the one "backing up" the affected main job, are dropped. Afterwards, no further copy jobs are released. Conversely, if the failure affected some copy job, all copy jobs are dropped.

The schedulability of each task τ_i must be guaranteed in both modes, also covering the possibility that a mode switch occurs *during* the busy period of τ_i. Therefore, τ_i must be provably schedulable under all the following scenarios, depending on what mode the system is in at the time of the task's absolute deadline:

- **Case 1:** Before any core fails (i.e. with the system in Mode S throughout the entire activation of τ_i).
- **Case 2:** In Mode D, triggered by a core failure affecting some higher priority task τ_k.
- **Case 3:** In Mode D, triggered by a core failure affecting τ_i itself.
- **Case 4:** In Mode D, triggered by a core failure affecting some lower priority task.

Case 1 can be covered by adapting the state-of-the-art analysis for global fixed-priority scheduling [17] and our model to each other, as we will describe. In Cases 2-4, we need to additionally account for the mode change (potentially also involving a reduction in the number of cores, if the core failure is permanent). But we note that Case 4 is dominated by Case 2, since lower-priority tasks cannot interfere with τ_i; and the reduction in the number of cores (in the case of permanent failure) is already captured by Case 2. In other words, all other things being equal, a failure of a higher-priority task τ_k (Case 2) causes both an immediate reduction in the number of cores *and* an increase in the workload by τ_k, which interferes with τ_i. By comparison, when a lower-priority task fails (Case 4), this only reduces the number of cores but it does not increase the higher-priority workload interfering with τ_i.

Hence, we only consider Cases 1-3 and use the superscripts $\varnothing$, $\not{k}$ and $\not{i}$ to differentiate among them in our notation. But first we briefly present the state-of-the-art analysis [17] for standard global fixed-priority scheduling, that we build upon.

B. The state-of-the art analysis by Guan et al. [17] for standard global fixed-priority scheduling

Under that analysis [17], τ_i is deemed schedulable if $R_i \leq D_i$, where R_i is an upper bound on its worst-case response time (WCRT), computed as

$$R_i = \begin{cases} C_i, & \text{if } |hp(i)| < m \\ C_i + I_i(R_i), & \text{otherwise} \end{cases} \tag{1}$$

where $I_i(R_i)$ is an upper bound on the interference that τ_i suffers from higher priority tasks during any interval of length R_i. This is computed as follows:

If a higher-priority task $\tau_j \in hp(i)$ has no carry-in workload within the busy-period of τ_i, its workload over an interval of length t is bounded by the expression

$$W_j^{NC}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \cdot C_j + [\![t \bmod T_j]\!]^{C_j} \tag{2}$$

where the term $t \bmod T_j \overset{\text{def}}{=} t - \left\lfloor \frac{t}{T_j} \right\rfloor$ (intuitively, the remainder of the division of t by T_j) corresponds to the length of the carry-out interval and the operator $[\![\cdot]\!]^{\cdot}_{\cdot}$ is defined as

$$[\![x]\!]_{\min}^{\max} \overset{\text{def}}{=} \begin{cases} \min & \text{if } x < \min \\ x & \text{if } \min \leq x \leq \max \\ \max & \text{if } x > \max \end{cases}$$

with the arguments $\min$ and $\max$ being optional and defaulting to $-\infty$ and $+\infty$ respectively, if omitted (Figure 2).

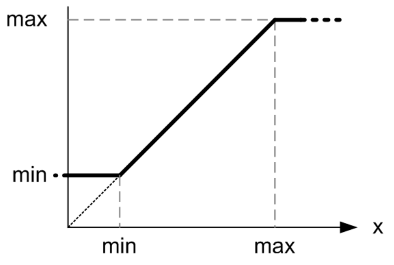


Fig. 2: Plot of $[\![x]\!]_{\min}^{\max}$ as a function of x.

However, in the general case, τ_j may also have carry-in workload. Then, its workload has the following upper-bound:

$$\begin{aligned} W_j^{CI}(t) = & \overbrace{\left\lfloor \frac{[\![t - C_j]\!]_0}{T_j} \right\rfloor \cdot C_j}^{\text{body}} + \overbrace{C_j}^{\text{carry-out}} \\ & + \underbrace{[\![[\![t - C_j]\!]_0 \bmod T_j - (T_j - R_j)]\!]_0^{C_j - 1}}_{\text{carry-in}} \end{aligned} \tag{3}$$

In Appendix A, we provide some intuition on how Guan et al. [17] derived these equations.

It is always the case that $W_j^{CI}(t) \geq W_j^{NC}(t)$. However, as Guan et al. [17] showed, improving on the work of Bertogna and Cirinei [8], at most $m-1$ tasks can have carry-in. Accordingly, an upper bound on the interference suffered by τ_i within the time interval R_i can then be computed as

$$\begin{aligned} I_i(R_i) = \Bigg\lfloor \frac{1}{m} \Big(& \sum_{j \in hp^{CI}(i)} [\![W_j^{CI}(R_i)]\!]^{R_i - C_i + 1} \\ & + \sum_{j \in hp^{NC}(i)} [\![W_j^{NC}(R_i)]\!]^{R_i - C_i + 1} \Big) \Bigg\rfloor \end{aligned} \tag{4}$$

where $hp^{CI}(i)$ is the subset, of cardinality $m-1$, of $hp(i)$ for which

$$\sum_{j \in hp^{CI}(i)} \left([\![W_j^{CI}(R_i)]\!]^{R_i - C_i + 1} - [\![W_j^{NC}(R_i)]\!]^{R_i - C_i + 1} \right)$$

is maximised; and $hp^{NC}(i) \overset{\text{def}}{=} hp(i) \setminus hp^{CI}(i)$.

Due to R_i appearing in both sides, Equation 1 is solved via a recurrence relation, as in uniprocessor WCRT analysis [3]. Note however that $hp^{CI}(i)$ and $hp^{NC}(i)$ need to be computed anew at every iteration.

C. Analysis for the failure-resilient model

1) Schedulability analysis for Mode S (Case 1): Our model differs from that of Guan et al. [17] mainly because of the copy jobs which, in the case of overlapping tasks, may concurrently exist with the corresponding main jobs. However, via a few key observations and transformations, we will adapt our model and the equations of Guan et al. [17] to each other.

Let $R_i^{\varnothing}$ denote an upper bound on the WCRT of the main job of τ_i under our model, under Case 1 (i.e. when the system is in Mode S).

Lemma 1: Under core-failure-resilient global fixed-priority scheduling with copy jobs, any copy job of task τ_j, whose corresponding main job is not directly affected by a core failure, executes for at most:
– zero time units, if τ_j is non-overlapping;
– no more than $\min(C_j, R_j^{\varnothing} - O_j)$, if τ_j is overlapping.

Proof: If τ_j is non-overlapping, it never even releases a copy job *unless* its corresponding main job has failed – which would contradict the initial assumption.

If τ_j is overlapping, no copy job by it can execute for more than C_j time units, by definition. But it cannot execute for more than $R_j^{\varnothing} - O_j$ time units either, because, according to the initial assumption, at most $R_j^{\varnothing} - O_j$ time units after the release of the copy job, its corresponding main job will have completed. And the copy job is terminated early at the same instant that its corresponding main job completes. ■

Remark 1: The copy jobs by an overlapping task τ_j have an inter-arrival time of T_j.

Proof: Follows directly from the fact that copy jobs are released at a fixed offset O_j, relative to the respective main jobs (which have an inter-arrival time of T_j), or not at all. ■

Lemma 1 and Remark 1 allow us to conveniently model, in Mode S, each overlapping task τ_j as two distinct tasks (main and copy), conforming to the semantics of classical scheduling (a single job per activation) and released at an offset O_j, with the main task τ_j having a WCET of C_i and the copy task τ_j' a WCET of $C_j' \overset{\text{def}}{=} \min(C_j, R_j^{\varnothing} - O_j)$ time units[2]. Figure 3 illustrates the relationship between copy offset O_j and C_j'.

By treating the main and the copy jobs as two independent tasks, we can then safely apply the existing analysis of Guan et al. [17] for global fixed-priority scheduled systems, at the cost of a bit of pessimism. For symmetry let us also model every non-overlapping task as a main/copy task pair, τ_j and τ_j', with the copy task having $C_j' = 0$.

[2]It is possible that, instead of a main job, it is the copy job of some overlapping task that is affected by a core failure. Yet, this case is always dominated by the case that the main job is the one wasted – simply because it is impossible for a copy job to have received more execution time than its corresponding main job, at the time of the failure. Hence, we only need to consider the main job being wasted.

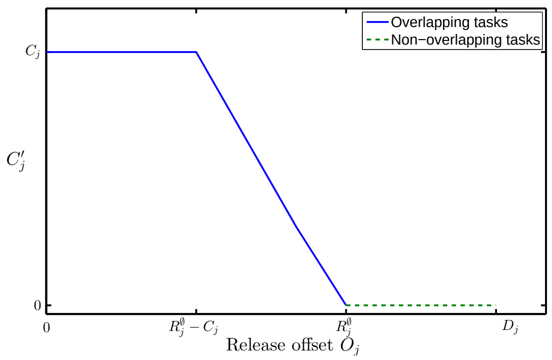


Fig. 3: The relationship between O_j and C_j'.

Then, a task τ_i is deemed schedulable under Case 1 if $R_i^{\varnothing} \leq D_i$ where

$$R_i^{\varnothing} = \begin{cases} C_i, & \text{if } |hp(i)| + |hp_{ov}(i)| < m \\ C_i + I_i^{\varnothing}(R_i^{\varnothing}), & \text{otherwise} \end{cases} \tag{5}$$

where $hp_{ov}(i) \subseteq hp(i)$ is the subset of higher-priority tasks that are overlapping.

To simplify the notation when computing $I_i^{\varnothing}$ let τ_{n+j} denote τ_j' and $hp^*(i) \overset{\text{def}}{=} \cup_{j \in hp(i)} \{\tau_j, \tau_j'\}$. (Since the following equations do not differentiate between main/copy tasks, this allows us to refer to the members of $hp^*(i)$ using a single non-ambiguous index.)

To compute $I_i^{\varnothing}$, we calculate $W_j^{\varnothing|NC}(t)$ and $W_j^{\varnothing|CI}(t)$, for each $j \in hp^*(i)$, similarly as before:

$$W_j^{\varnothing|NC}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \cdot C_j + [\![t \bmod T_j]\!]^{C_j} \tag{6}$$

$$W_j^{\varnothing|CI}(t) = \overbrace{\left\lfloor \frac{[\![t - C_j]\!]_0}{T_j} \right\rfloor \cdot C_j}^{\text{body}} + \overbrace{C_j}^{\text{carry-out}} \\ + \underbrace{[\![[\![t - C_j]\!]_0 \bmod T_j - (T_j - R_j^{\varnothing})]\!]_0^{C_j - 1}}_{\text{carry-in}} \tag{7}$$

Specifically for tasks in $hp^*(i)$ that are copy tasks of the original task set, we need to define $R^{\varnothing}$, for use in the above equations. Since a copy job, under Case 1 (i.e. in Mode S), is terminated short of completion, upon the completion of its corresponding main job, then as response time of a copy task τ_j' (also denoted by τ_{n+j}), we can consider the quantity $R_{n+j}^{\varnothing} \overset{\text{def}}{=} R_j^{\varnothing} - O_j$. As for non-overlapping tasks, the workload of their copies is, by definition for Case 1, equal to zero.

Then, reasoning similarly as before

$$I_i^{\varnothing}(R_i^{\varnothing}) = \left\lfloor \frac{1}{m}\Big(\sum_{j \in hp^{\varnothing|CI}(i)} [\![W_j^{\varnothing|CI}(R_i^{\varnothing})]\!]^{R_i^{\varnothing}-C_i+1} + \sum_{j \in hp^{\varnothing|NC}(i)} [\![W_j^{\varnothing|NC}(R_i^{\varnothing})]\!]^{R_i^{\varnothing}-C_i+1}\Big)\right\rfloor \quad (8)$$

where $hp^{\varnothing|CI}(i)$ is the subset, of cardinality $m-1$, of $hp^{*}(i)$ for which

$$\sum_{j \in hp^{\varnothing|CI}(i)} \Big([\![W_j^{\varnothing|CI}(R_i^{\varnothing})]\!]^{R_i^{\varnothing}-C_i+1} - [\![W_j^{\varnothing|NC}(R_i^{\varnothing})]\!]^{R_i^{\varnothing}-C_i+1} \Big)$$

is maximised; and $hp^{\varnothing|NC}(i) \overset{\text{def}}{=} hp^{*}(i) \setminus hp^{\varnothing|CI}(i)$.

Similarly to the original analysis [17], $hp^{\varnothing|CI}(i)$ and $hp^{\varnothing|NC}(i)$ need to be computed anew at every iteration of the recurrence relation when solving the response time equation.

2) Schedulability analysis for the case of a core failure affecting some higher priority task τ_k (Case 2): A core failure by affecting some $\tau_k \in hp(i)$ creates additional interference on τ_i in the short term. In the worst case, when τ_k fails just before completion, its main and copy jobs combined could execute for up to $2C_k - \epsilon$ time units within its period of T_k (with ϵ arbitrarily small). Additionally, in the case of a permanent core failure, the system would be left with one core less ($m' = m-1$) with which to process the workload, making it even harder to meet deadlines. On the other hand, dropping any other copy jobs currently executing and no longer releasing any of them in the future, after a core failure, eases up on the workload, especially on the longer term. In practice, this means that Case 2 has to be analysed separately for each higher-priority task and also that schedulability cannot be inferred from Case 1 either.

Let us now try to bound the interference onto τ_i both from τ_k (the task affected by the failure) and also from every other higher-priority task.

Higher-priority task activations whose deadlines fall before the core failure: This includes all activations whose deadlines fall earlier than the time instant of a core failure. Given that these activations complete at a time when the system is still in Mode S, our earlier reasoning for Case 1 applies. Hence, the respective main and copy jobs of the same task τ_j can be modelled as originating from distinct tasks τ_j and τ_j', with the latter's execution time being

$$C_j' = \begin{cases} 0 & \text{if } \tau_j \text{ is non-overlapping} \\ \min(C_j, R_j^{\varnothing} - O_j) & \text{if } \tau_j \text{ is overlapping} \end{cases} \quad (9)$$

Note that the above derivation requires the value of $R_j^{\varnothing}$ (from Case 1) to have been already computed for each higher-priority task τ_j. This has implications for the order in which the task WCRTs need to be computed for the different cases (1-3) under consideration. We revisit this subject in Sections IV-D and IV-E.

Again each copy task τ_j' is mapped to a task τ_{n+j}, for ease of referencing in equations, and $hp^{\cancel{k}}(i) \overset{\text{def}}{=} \cup_{j \in hp(i)}\{\tau_j, \tau_j'\}$.

Higher-priority task activations whose deadlines fall after the core failure: When considering the higher-priority task activations whose deadlines fall after the core failure, we need to distinguish between the task τ_k directly affected by the failure and every other higher-priority task $\tau_j \neq \tau_k$.

For some higher-priority task $\tau_j \neq \tau_k$, we note that all activations released after the time instant of failure t_f exert workload of up to C_j time units (i.e. from their main job only), since copy jobs are no longer released. However, activations released before t_f have workload both from the main and the copy job, in the general case. Then, it is safe to use Equation 9 to upper-bound for the respective per-job workloads. To further simplify the analysis (at the cost of some additional pessimism) let us use Equation 9 to also bound the workloads of jobs released after t_f, for every higher-priority job not directly affected by the failure. This approach allows us to obtain a safe upper-bound on the workload, without the need to identify the time instant t_f, which leads to the worst-case scenario.

As for the task τ_k that was directly affected by the failure, the same reasoning as for $\tau_j \neq \tau_k$ applies, albeit with one crucial difference. The last interfering copy job by τ_k, i.e. the one that completes filling in for the main job that was terminated early by the failure, may execute up to the entire C_k – not C_k', if smaller. This effect can be incorporated in the modelling of workload as follows:
– If τ_k' (the copy task of τ_k) has carry-in workload, we set in the equations its carry-out workload equal to C_k, not C_k'. Note that this may have a "knock-on" effect on the length of the body and carry-in intervals.
– If it has no carry-in workload, exactly one of the jobs contributing to the workload within the interval in consideration is modelled as having execution requirement of C_k and all other jobs are modelled with an execution requirement of C_k'. To simplify the derivation, potentially at the cost of some pessimism, we shift the execution requirement C_k to the instance at the start of the "body".

Based on the above reasoning we get:

For every $\tau_j \in hp^{*}(i)$, $j \notin \{k, n+k\}$:

$$W_j^{\cancel{k}|NC}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \cdot C_j + [\![t \bmod T_j]\!]^{C_j} \quad (10)$$

$$W_j^{\cancel{k}|CI}(t) = \overbrace{\left\lfloor \frac{[\![t - C_j]\!]_0}{T_j} \right\rfloor \cdot C_j}^{\text{body}} + \overbrace{C_j}^{\text{carry-out}} + \underbrace{[\![[\![t - C_j]\!]_0 \bmod T_j - (T_j - R_j^{\varnothing})]\!]_0^{C_j - 1}}_{\text{carry-in}} \quad (11)$$

Analogously as in Case 1, for each $\tau_{n+j} \in hp^{*}(i)$, (i.e. the copy task of $\tau_j \in hp(i)$), we assume $R_{n+j}^{\varnothing} \overset{\text{def}}{=} R_j^{\varnothing} - O_j$, if τ_j is overlapping.

For the main task τ_k:

$$W_k^{\not{k}|NC}(t) = \left\lfloor \frac{t}{T_k} \right\rfloor \cdot C_k + \llbracket t \bmod T_k \rrbracket^{C_k} \tag{12}$$

$$\begin{aligned} W_k^{\not{k}|CI}(t) = {} & \overbrace{\left\lfloor \frac{\llbracket t - C_k \rrbracket_0}{T_k} \right\rfloor \cdot C_k}^{\text{body}} + \overbrace{C_k}^{\text{carry-out}} \\ & + \underbrace{\llbracket \llbracket t - C_k \rrbracket_0 \bmod T_k - (T_k - R_k^{\varnothing}) \rrbracket_0^{C_k - 1}}_{\text{carry-in}} \end{aligned} \tag{13}$$

For its copy, denoted as τ'_k or equivalently as τ_{n+k}:

$$\begin{aligned} W_{n+k}^{\not{k}|NC}(t) = {} & \llbracket t \rrbracket_0^{C_k} + \left\lfloor \tfrac{t - T_k}{T_k} \right\rfloor_0 \cdot C'_k \\ & + \Big\llbracket \llbracket t - T_k \rrbracket_0 \bmod T_k \Big\rrbracket^{C'_k} \end{aligned} \tag{14}$$

$$\begin{aligned} W_{n+k}^{\not{k}|CI}(t) = {} & \overbrace{\left\lfloor \frac{\llbracket t - \mathbf{C_k} \rrbracket_0}{T_k} \right\rfloor \cdot C'_k}^{\text{body}} + \overbrace{\mathbf{C_k}}^{\text{carry-out}} \\ & + \underbrace{\llbracket \llbracket t - \mathbf{C_k} \rrbracket_0 \bmod T_k - (T_k - R_{n+k}^{\varnothing}) \rrbracket_0^{C'_k - 1}}_{\text{carry-in}} \end{aligned} \tag{15}$$

Note the use of C_k and C'_k in Equation 15. Specifically, for the jobs of τ'_k contributing to the carry-out part of the workload, the entire C_k is assumed, which is necessary, due to the failure that affected τ_k. Conversely, for the jobs constituting the body and the carry-in parts, only C'_k is used.

Putting this all together, we have

$$\begin{aligned} I_i^{\not{k}}(R_i^{\not{k}}) = \Bigg\lfloor \frac{1}{m'} \Big(& \sum_{j \in hp^{\not{k}|CI}(i)} \llbracket W_j^{\not{k}|CI}(R_i^{\not{k}}) \rrbracket^{R_i^{\not{k}} - C_i + 1} \\ & + \sum_{j \in hp^{\not{k}|NC}(i)} \llbracket W_j^{\not{k}|NC}(R_i^{\not{k}}) \rrbracket^{R_i^{\not{k}} - C_i + 1} \Big) \Bigg\rfloor \end{aligned} \tag{16}$$

where $hp^{\not{k}|CI}(i)$ is the subset, of cardinality $m-1$, of $hp^*(i)$ for which

$$\sum_{j \in hp^{\not{k}|CI}(i)} \Bigg(\llbracket W_j^{\not{k}|CI}(R_i^{\not{k}}) \rrbracket^{R_i^{\not{k}} - C_i + 1} - \llbracket W_j^{\not{k}|NC}(R_i^{\not{k}}) \rrbracket^{R_i^{\not{k}} - C_i + 1} \Bigg)$$

is maximised; and $hp^{\not{k}|NC}(i) \overset{\text{def}}{=} hp^*(i) \setminus hp^{\not{k}|CI}(i)$.

Accordingly

$$R_i^{\not{k}} = \begin{cases} C_i, & \text{if } |hp(i)| + |hp_{ov}(i)| < m' \\ C_i + I_i^{\not{k}}(R_i^{\not{k}}) & \text{otherwise} \end{cases} \tag{17}$$

Note the use of term m' rather than m in Equations 16 and 17. The symbol m' refers to the number of available cores after the failure, namely

$$m' = \begin{cases} m, & \text{in case of a non-permanent core failure} \\ m-1, & \text{in case of a permanent core failure} \end{cases}$$

We pessimistically assume that the number of usable cores is m' throughout the activation of τ_i in the WCRT equations, to be on the safe side without having to identify the instant of failure t_f leading to the worst-case scenario. Similarly, although the number of higher-priority tasks with carry-in can be at most $m'-1$ in Mode D, consistent with the reasoning of Guan et al. [17], we pessimistically assume $m-1$ such tasks throughout (i.e. in the definition of $hp^{\not{k}|CI}(i)$), in order to be on the safe side.

3) Schedulability analysis for the case of a core failure affecting the task τ_i under analysis (Case 3): This case concerns the schedulability of some task τ_i, when its main job is terminated early, due to a core failure. Then its corresponding copy job must complete within the same absolute deadline. The condition for schedulability is:

$$R_{n+i}^{\not{i}} \leq \begin{cases} D_i - O_i, & \text{if } \tau_i \text{ is overlapping} \\ D_i - R_i^{\varnothing}, & \text{if } \tau_i \text{ is non-overlapping} \end{cases} \tag{18}$$

where $R_{n+i}^{\not{i}}$ is the worst-case response time of a job by the copy task τ_{n+i} (equivalently denoted as τ'_i) assuming that its corresponding main job was directly affected by the core failure. The latest that this copy job can be released is O_i time units after the release of its corresponding main job, if τ_i is overlapping, or respectively, $R_i^{\varnothing}$ time units, if non-overlapping. This reduces accordingly the effective relative deadline for the copy job, leading to the condition of Equation 18.

Bounding the workloads by higher-priority jobs is analogous to the previous cases. For every $\tau_j \in hp^*(i)$:

$$W_j^{\not{i}|NC}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \cdot C_j + \llbracket t \bmod T_j \rrbracket^{C_j} \tag{19}$$

$$\begin{aligned} W_j^{\not{i}|CI}(t) = {} & \overbrace{\left\lfloor \frac{\llbracket t - C_j \rrbracket_0}{T_j} \right\rfloor \cdot C_j}^{\text{body}} + \overbrace{C_j}^{\text{carry-out}} \\ & + \underbrace{\llbracket \llbracket t - C_j \rrbracket_0 \bmod T_j - (T_j - R_j^{\varnothing}) \rrbracket_0^{C_j - 1}}_{\text{carry-in}} \end{aligned} \tag{20}$$

However, we also need to consider the potential interference from the main job of τ_i.

Lemma 2: The interfering workload that some copy job τ'_i can suffer by its corresponding main job, assuming that no higher-priority task τ_j has been affected by a core failure, is at most C'_i.

Proof: If τ_i is non-overlapping, its main and copy jobs can never concurrently exist. Otherwise, if τ_i is overlapping, the main can execute for at most $\min(C_i, R_i^{\varnothing} - O_i)$ time units after the release of its copy before it completes (or is terminated due to failure). Given the definition of C'_i (Equation 9), the claim then holds in both cases. ■

Lemma 2 forms an upper-bound on interfering workload upon τ'_i from any job by τ_i including the one affected by the core failure. Additionally, we know that at most one job by τ_i (i.e. the one terminated early due to the failure) interferes with the copy job τ'_i under analysis. Therefore the interfering workload from τ_i is C'_i and

$$I^{\not{i}}_{n+i}(R^{\not{i}}_{n+i}) = \Bigg\lfloor \frac{1}{m'} \Bigg(\sum_{j \in hp^{\not{i}|CI}(i)} [\![W_j^{\not{i}|CI}(R^{\not{i}}_{n+i})]\!]^{R^{\not{i}}_{n+i} - C_i + 1} + \sum_{j \in hp^{\not{i}|NC}(i)} [\![W_j^{\not{i}|NC}(R^{\not{i}}_{n+i})]\!]^{R^{\not{i}}_{n+i} - C_i + 1} + \mathbf{C'_i} \Bigg) \Bigg\rfloor \quad (21)$$

where $hp^{\not{k}|CI}(i)$ and $hp^{\not{k}|NC}(i)$ are computed analogously as before. Note that the copy job (τ'_i), in this case, needs to execute for the entire C_i – not just C'_i, because it is not terminated early. Thus:

$$R^{\not{i}}_{n+i} = \begin{cases} C_i, & \text{if } |hp(i)| + |hp_{ov}(i)| + \text{is\_ov(i)} < m' \\ C_i + I^{\not{i}}_{n+i}(R^{\not{i}}_{n+i}) & \text{otherwise} \end{cases} \quad (22)$$

where is_ov(i) $=$ 1 if and only if τ_i is overlapping; otherwise it is zero.

D. Offset selection for copy tasks

To test the schedulability of some τ_i, we need as inputs the $R^{\varnothing}_j$ of each $\tau_j \in hp(i)$ and knowledge of whether that τ_j is overlapping (and if so, its copy job offset O_j). This means that tasks have to be analysed for schedulability in a top-down priority order, assuming a given priority assignment (which itself is being tested for feasibility). By inspecting the WCRT equations, we also note that, $R^{\varnothing}_i$ of each task τ_i serves as input for the calculation of $R^{\not{k}}_i$s, $\forall k \in hp(i)$ and also $R^{\not{i}}_{n+i}$; so it has to be computed before those.

It is only at the stage of testing the schedulability of τ'_i (Case 3) that we need to consider whether or not τ_i is overlapping – and if so, what its copy offset O_i is. For the purposes of schedulability testing, a non-overlapping τ_i can be equivalently modelled as having a fixed offset of $O_i = R^{\varnothing}_i$. Given that the selection of O_i entails a tradeoff, as earlier discussed, its value is thus best decided in tandem with the schedulability testing at Case 3 for the task in consideration. A task should be non-overlapping, if possible, to avoid interference from redundant execution onto lower priority tasks. But if a non-overlapping arrangement is not schedulable, then it is desirable to set O_i to the highest value that makes τ' schedulable, in order to minimise the overlap-related interference onto lower-priority tasks. We optimally identify this value using the algorithm of Figure 5, as proven by Theorem 1.

Theorem 1: The algorithm of Figure 5 optimally selects the copy offset O_i.

Proof: If the task is schedulable without overlap, then the loop is never entered and SUCCESS is declared. But if it is

```
1.   int calculate_O($\tau_i$) //in tandem with R$^{\not{i}}_{n+i}$
2.   {O$_i$:=R$^{\varnothing}_i$; //initialisation
3.    calculate R$^{\not{i}}_{n+i}$;
4.    while (O$_i$+R$^{\not{i}}_{n+i}$ >D$_i$)
5.    {O$_i$:=D$_i$-R$^{\not{i}}_{n+i}$; //adjusting offset
6.     if (O$_i$<0) return FAILURE;
7.     calculate R$^{\not{i}}_{n+i}$; //using new O$_i$
8.    }
9.    return SUCCESS;
10.  }
```

Fig. 5: Calculation of O_i and $R^{\not{i}}_{n+i}$ in tandem.

unschedulable without overlap, then let O^v denote the value that O_i is updated to within the v^{th} loop iteration and R^v denote the $R^{\not{i}}_{n+i}$ calculated using O^v during the same iteration. O^{v-1} and R^{v-1} then refer to the respective previous values.

Then, in line 5, $O^{v+1} = O^v - ((O^v + R^v) - D_i)$, i.e., the offset is decreased by the amount of time that the deadline is exceeded. And since, by inspection $R^{\not{i}}_{n+i}$ cannot decrease when O_i decreases, this means that all offsets in the range $[O^{v+1},\ O^v)$ would have been infeasible. Hence, each iteration disqualifies an infeasible subrange of $[0,\ R^{\varnothing}_i]$, from right to left. And if a feasible offset is identified, then this will be the greatest such offset. Moreover, because task parameters are integers, after a finite number of iterations, either a feasible offset is found or the entire range for the offset $[0,\ R^{\varnothing}_i]$ is found infeasible (line 6). ■

Note that, although all candidate offsets greater than the one derived (in case of success) by the algorithm of Figure 5 are provably infeasible (according to Theorem 1), not all smaller offsets will be feasible in the general case, because for some offset ranges, $R^{\not{i}}_{n+i}$ may increase more than the amount by which the offset is decreased. Therefore, the optimal offset could not have been identified, e.g., by simpler approaches such as iterative binary (dichotomic) search. Applying binary search would either result in false negatives regarding the schedulability of the copy task or in suboptimal offsets (i.e., unnecessarily big overlap).

E. Priority assignment

For sporadic global fixed-priority scheduling, no exact schedulability tests are known. Therefore, it makes sense to speak of an optimal priority assignment scheme only in the context of a given, sufficient schedulability test [14]. Then a priority assignment scheme is optimal in the context of a test T if it always finds a priority assignment for which the task is proven schedulable using the test T, if such an assignment exists at all. An obvious but often intractable ($O(n!)$) optimal priority assignment scheme, for any test T is exhaustive enumeration.

Davis and Burns [14] demonstrated that Audsley's OPA [2], a bottom-up scheme with pseudo-polynomial $O(n^2)$ complexity, is optimal in the above sense for global scheduling if the schedulability test fulfils some conditions. One of those requirements is that the schedulability of a task, using the test in consideration, must depend only on the set of higher-priority tasks – but not their relative priorities. Our schedulability analysis does not meet this condition because, as discussed

```
1.   int test_task_schedulability(int i, int p)
2.   {if !($\tau_i$ is not schedulable at priority p when no core fails) return FAILURE;
3.    if !($\tau_i$ is not schedulable at priority p when a core fails and $\tau_k$ is killed, $\forall \tau_k \in hp(i)$) return FAILURE;
4.    if !($\tau_i$ is not schedulable at priority p when a core fails and $\tau_i$ itself is killed) return FAILURE;
5.    // The test of the above line also computes $O_i$, in the process, if successful.

6.    return SUCCESS;
7.   }
```

Fig. 4: Pseudocode for testing the core-failure-resilient schedulability of a task τ_i at a priority level p. This assumes that the set $hp(i)$ of higher-priority tasks, and their priorities and copy jobs offets are specified, and also and that every $\tau_j \in hp(i)$ is core-failure-resiliently schedulable, according to the same test.

earlier, to test the schedulability of τ_i at a given priority (Figure 4) it requires $R_j^{\varnothing}, \forall j \in hp^*(i)$ – i.e. it depends on the relative priorities of higher-priority tasks.[3] Hence the most practical *optimal* priority assignment scheme for our test is top-down, branch-and-bound enumeration, which is still intractable in the general case ($O(n!)$).

Fortunately, our experiments showed that suboptimal tractable heuristics exist that in practice perform very close to optimal. Namely, the DkC scheme [1], which assigns priorities (high to low) in order of increasing $D_i - k \cdot C_i$, where k is a tunable constant. In particular, a value of $k = 1.1$, which roughly corresponds to a Slack Monotonic priority assignment, seemed to work best and significantly outperformed Deadline Monotonic (DM), in the context of our scheduling approach. Trying a few different DkC priority assignments, corresponding to different values of k, until success, barely underperformed the optimal branch-and-bound exhaustive enumeration. Figure 8 in Appendix B shows how (i) DM, (ii) DkC with $k = 1.1$, (iii) DkC $\forall\ k \in \{0, 0.1, \ldots 2.0\}$ and (iv) the optimal priority assignment perform, in a small-scale experiment, with $m = 4$ processors and only $n = 7$ tasks. Note that the relatively small values for the parameters m and n were deliberately chosen, so that the optimal branch-and-bound enumeration terminates within a reasonable computation time. For further explanations, please refer to Section VI and Appendix B.

V. Practical considerations

In this chapter we discuss some challenges related to the implementation of the process model assumed by our approach, and offer some suggestions on how to overcome them:

Race conditions/synchronisation hazards resulting of the coexistence of jobs (main and copy) by the same task: In this work we assumed that tasks share no resources with each other; this is an unrealistic assumption but we plan to remove it in our future work by leveraging the existing results on resource sharing protocols and introducing appropriate safe blocking terms to the schedulability analysis. However, specifically in the case of overlapping tasks, another challenge arises. When the main job and the copy job of *the same* task coexist in the system, they may both access resources for which the programmer never expected any concurrent access by different processes. Namely, if the development was done obliviously to the fact that our scheduling arrangement, with job copies, would be employed, for resilience against core failures. For this reason, we assume a task structure design whereunder (i) upon its release each job copies its input variables into its private address space; (ii) during its execution, it only updates/manipulates its own local variables; and (iii) it only produces its output (i.e. as I/O, or by updating persisten variables in the system memory) as a final commit stage, before completing. Such a model might indeed cover many applications (for example: read sensor value/do computation/emit output to control actuator) but not all. Additionally, some buffering mechanism which stores dequeued inputs for the copy job might be needed. On the other hand, for tasks which require I/O halfway through their execution, the aforementioned approach may not be practical or enforceable. But it may still be possible to accommodate such tasks, i.e. via appropriate priority assignment, which ensures that they will be non-overlapping.

Transient/permanent core failure detection: We assumed earlier that core failures are detected instantaneously. However, in practice, even with hardware support, failure detection will still require some degree of software support and this means that it will have some latency and associated scheduling overheads. These would have to be incorporated into the model and the schedulability analysis, taking into account exactly how the failure detection facility is implemented.

Launching and early termination of copy jobs: Obviously, a software facility is required for tracking the arrivals of the main jobs and launching the copy jobs at the appropriate offsets. However, the same facility should keep track of job completions in order to immediately also terminate short of completion the corresponding job copies. Note that it is possible (i.e., in case of different control flow) for an overlapping copy job to complete before its corresponding main job, in which case it is then the main job that should be terminated short of completion.

VI. Evaluation

We evaluate our approach by testing the schedulability of thousands of task sets, generated using UUnifast-Discard [14] (extension of [9])[4]. We plot the scheduling success ratios as a function of the system utilisation ($U_s \overset{\text{def}}{=} \frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i}$) for:

OursH/OursS: Our approach, assuming permanent/transient core failures, respectively.
Guan: The state-of-the-art test [17] for global fixed priorities without provision for fault resilience, purely for reference.
Dupl-Part-FP: A fault-resilient partitioned arrangement, with

[3]Modifying the test to use D_j instead of $R_j^{\varnothing}$ as input would make OPA optimal in its context but would be counter-productive due to the resulting pessimism in the WCRT derivations.

[4]Each percentage point represents 200 implicit-deadline (D=T) task sets. Task periods were chosen uniformly over [30000, 100000] μsec.

full task duplication, that assigns the two copies of the same task to different cores, via Best-Fit bin-packing[5], and uses Deadline Monotonic (DM) priority assignment (optimal, on each core).
Dupl-Part-EDF: The same, but using partitioned EDF.

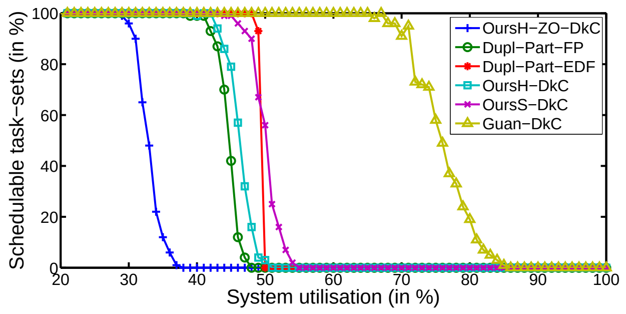
Different configurations of the above approaches:
ZO: Forcing full overlap for all tasks, i.e., $O_i = 0,\ \forall \tau'_i$.
DkC: Using DkC priority assignment, and trying all $k \in \{0, 0.1, \ldots, 1.9, 2\}$ until success.

Figure 6a contains the plots for systems with $m = 8$ cores and sets of $n = 16$ tasks. The schedulability improvement from optimally picking copy task offsets is considerable. OursH-DkC outperforms the duplicated partitioned fixed-priority arrangement and in case of transient core failures (OursS-DkC) the lead increases, as expected. In the experiment of Figure 6b (n=40), we note that the Ours-* curves improve, whereas for the other curves for fixed priorities, this is not the case or even a slight performance deteriotation is noted. The trend is amplified for n=80 (Figure 6c), which shows that our approach tends to be more efficient, for higher n/m. A higher n/m ratio means smaller average task utilisations, which one might expect would improve schedulability for all approaches, since it would mean smaller bin-packing-related fragmentation on average. However, no such effect occurs for the approaches with full duplication, because the average processor utilisation is twice the nominal system utilisation, so there is little room for improvement anyway. Additionally, the subset of tasks that never suffer any interference (i.e., because they are always guarranteed a processor due to their high priority) becomes a smaller fraction of the overall task set, for higher n/m; this increases the potential pessimism in the schedulability test, which explains our observations. In the case of OursH-DkC and OursS-DkC however, the performance improves for higher n/m, because the smaller per-task average utilisation allows more tasks to be accommodated without overlap. The scheduling performace gains from the low overlap can be assessed by comparing with OursH-ZO (the naive approach with full overlap), whose performace does not improve with higher n/m. The results for $m = 16$ cores were similar, so we do not include them.

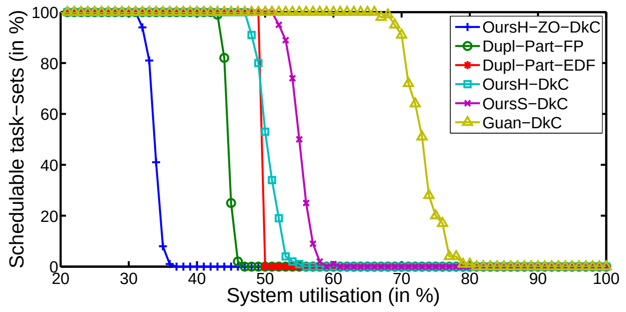
Dupl-Part-EDF outperforms our approach for task sets with lower n/m, which we attribute to the better scheduling potential associated with EDF, but this reverses for higher n/m. This reversal occurs, because, as noted earlier, the amount of redundant execution under our approach tends to decrease when the average task utilisation is smaller. Additionally, there is no room for better performance by Dupl-Part-EDF anyway, because neither this nor any other approach that employs full task duplication is capable of scheduling *any* task set with utilisation above half the system capacity, hence the sharp cutoff at 50%.

Next, we inspect the scheduling arrangements resulting from the application of our approach. We consider $m = 8$ cores, implicit-deadline tasks and permanent core failures. Figure 7a plots the average "effective" utilisation ($U^* \overset{\text{def}}{=} (1/m)\sum_{i=1}^{n}(C_i + C'_i)/T_i$) in Mode S for *schedulable* tasks sets, according to the nominal system utilisation U_s. Higher U^*/U_s indicates more overlap. We note that hardly any overlap

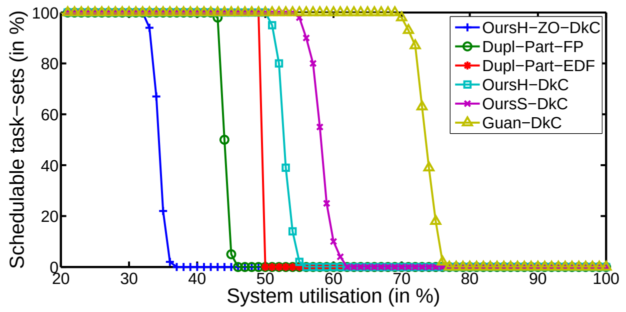
[5]In our experiments it outperforms First-Fit.



(a) $m = 8$, $n = 16$



(b) $m = 8$, $n = 40$



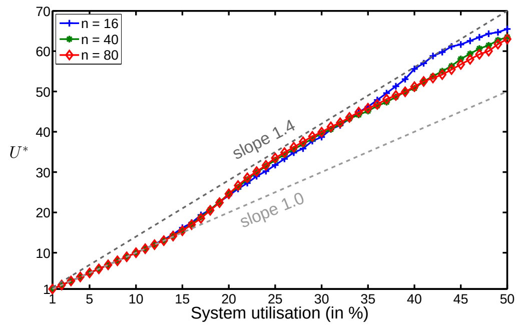
(c) $m = 8$, $n = 80$

Fig. 6: Experimental evaluation (part 1 of 2)

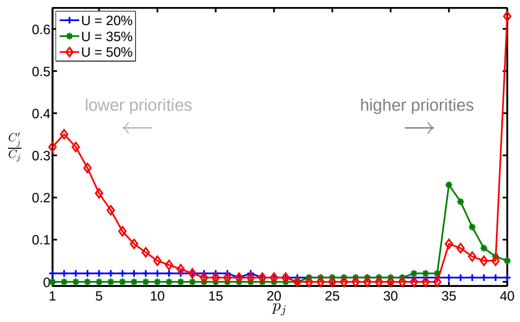
is needed to schedule low-utilisation task sets. This means that there is capacity for additional background soft-real time tasks without resilience guarantees. But even for high utilisations, the additional utilisation taken up by copies ($U^* - U_s$), does not exceed 40% of U_s on average.

Figure 7b, assuming $n = 40$ tasks plots the average degree of overlap C'_j/C_j according to the task priority. Three different system utilisations were considered. We note that middle-priority tasks barely require any overlap to be schedulable. For higher-utilised systems there may be some overlap in the higher or lower spectrum. High-priority tasks with overlap are mostly tasks that *must* have overlap (i.e., tasks with u_j>0.5), but which also can tolerate very little interference. Obviously, such tasks are rarer in lower-utilisation task sets (line $U = 20\%$). In any case, DkC does a good job at assigning high priorities to those. At the other end, it is natural that overlap tends to be higher at lower priorities, because these tasks suffer more interference and their copies need a commensurate "head start" (small O_j) to complete on time. Moreover, overlap at higher priorities tends to penalise schedulability because the additional interference is exerted onto more lower-priority tasks, in a cascade effect. Therefore, an efficient priority assignment would intuitively "shift" the overlap to the lower priorities. By inspection, DkC does that.

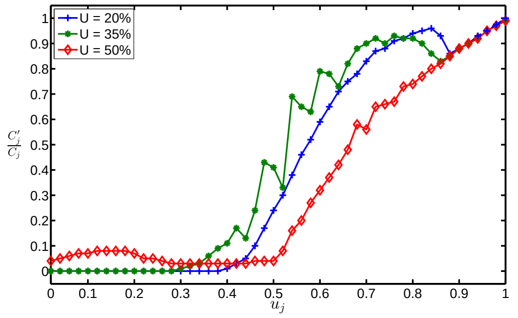
Figure 7c, examines the degree of overlap according to the task utilisation. By inspection, lower-utilisation tasks tend to require little overlap. Past 40%, the utilisation correlates positively with overlap. This behaviour matches our observations for the previous two graphs, especially with respect to the priority assignment output by DkC, and also explains why an approximate Slack Monotonic usually performs well.



(a) Effective utilisation $U^* \overset{\text{def}}{=} \sum_{i=1}^{n} \frac{C_i + C_i'}{T_i}$ vs nominal utilisation.



(b) Degree of overlap $\frac{C_j'}{C_j}$ according to the task priority.



(c) Degree of overlap $\frac{C_j'}{C_j}$ according to task utilisation $u_j \overset{\text{def}}{=} \frac{C_j}{T_j}$.

Fig. 7: Experimental evaluation (part 2 of 2)

VII. Conclusion

We introduced a new form of schedulability guarantees (for surviving a transient or permanent core failure), a global fixed-priority-based scheduling arrangement for achieving them and novel analysis for their derivation. The preliminary evaluation indicates the efficiency of our approach. Many directions for future work exist, such as: (i) to consider a global EDF policy, (ii) to augment the model with arbitrary deadlines and shared resources (including surviving a failure *during* the resource access), (iii) to reduce the analysis pessimism, and (iv) to extend the approach, so as to make it also applicable to scenarios with deferred (delayed) fault detection.

References

[1] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Proc. RTCSA*, 2000.

[2] N. Audsley. On priority assignment in fixed priority scheduling. *Inf. Proc. Letters*, 79(1):39–44, 2001.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[4] T. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 2006.

[5] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proc. RTSS*, 2007.

[6] S. Baruah and T. Baker. Schedulability analysis of global edf. *Real-Time Systems*, 2008.

[7] S. Baruah and Z. Guo. Mixed-criticality scheduling upon varying-speed processors. In *Proc. RTSS*, 2013.

[8] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proc. RTSS*, 2007.

[9] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2009.

[10] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Proc. RTSS*, 2009.

[11] A. Burns and R. Davis. Mixed criticality systems: A review. TR. Computer Science, U. of York, UK, 2013.

[12] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proc. ECRTS*, 2007.

[13] M. Cirinei, E. Bini, G. Lipari, and A. Ferrari. A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors. In *Proc. IPDPS*, 2007.

[14] R. I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Proc. RTSS*, 2009.

[15] Federal Aviation Authority. CAST-32: Multi-core processors. https://www.faa.gov/, 2014.

[16] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerant scheduling on a hard real-time multiprocessor system. In *Proc. IPPS*, 1994.

[17] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Proc. 30th RTSS*, 2009.

[18] A. Gujarati, F. Cerqueira, and B. Brandenburg. Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities. In *Proc. ECRTS*, 2013.

[19] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proc. ECRTS*, 2009.

[20] J. Lee and K. G. Shin. Schedulability analysis for a mode transition in real-time multi-core systems. In *Proc. RTSS*, 2013.

[21] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[22] L. Lundberg. Multiprocessor scheduling of age constraint processes. In *Proc. 5th RTCSA*, 1998.

[23] B. Nikolić and S. M. Petters. Towards network-on-chip agreement protocols. In *Proc. EMSOFT*, 2012.

[24] R. M. Pathan. Fault-tolerant real-time scheduling using chip multiprocessors. In *Proc. Suppl. vol. EDCC*, 2008.

[25] R. M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4):509–547, 2014.

[26] Y. Sun, G. Lipari, N. Guan, and W. Yi. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *Proc. RTCSA*, 2014.

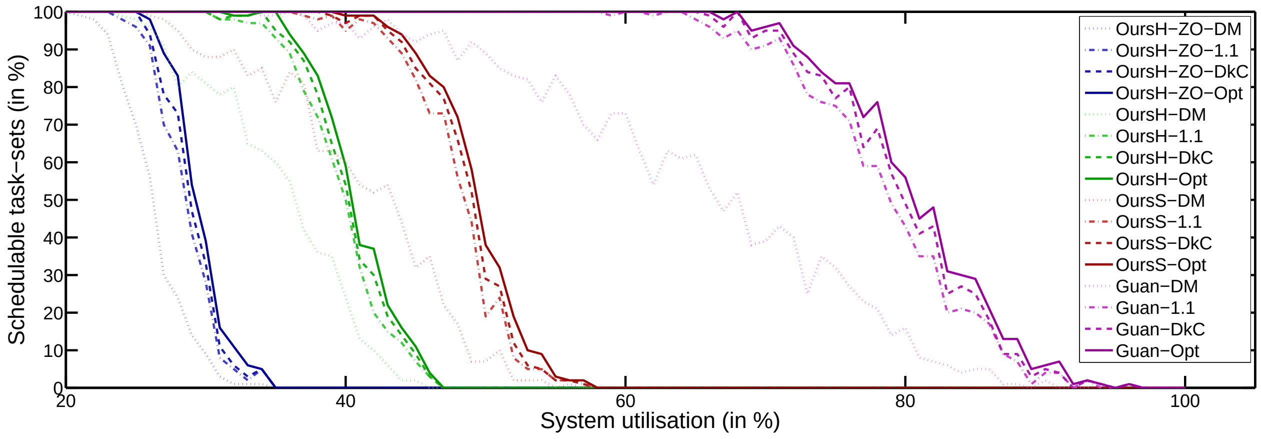


Fig. 8: Performance of different priority assignment schemes ($m = 4$, $n = 7$)

[27] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. RTSS*, 2007.

APPENDIX

A. Computation of interfering workload by Bertogna and Cirinei [8] and Guan et al. [17]

For rigorous derivation of the WCRT equations by Guan et al. [17], that extended the previous work of Bertogna and Cirinei [8], we refer the reader to their respective papers. Here we only provide intuitive explanations.

Carry-in workload by some task, within a given time window, is workload from jobs released earlier than the start of the time window in consideration.

For the case where there is no carry-in, the worst-case scenario, in terms of workload by some higher-priority task τ_j, resembles the "critical instant" which is well-known from uniprocessor scheduling (Figure 9(a)).

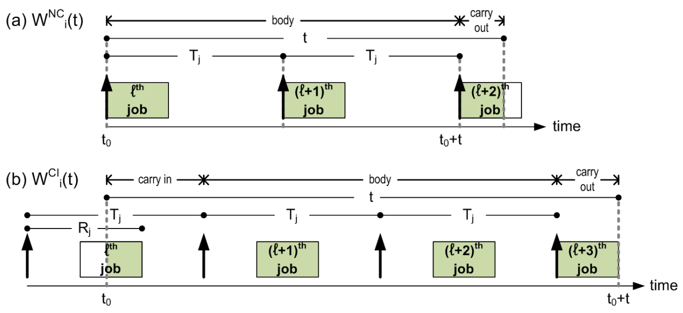


Fig. 9: Example to help understand computation of carry-ins

But, as Bertogna and Cirinei [8] showed, on globally-scheduled multiprocessors, this is not the worst-case scenario in the general case. Rather, the correctly identified worst-case scenario, in terms of workload by τ_j may involve carry-in workload. Intuitively, the corresponding execution pattern (Figure 9(b)) is created by the inverting/"flipping" right-to-left the workload pattern of Figure 9(a) in the following way:

Carry-out: The last job by τ_j contributing to the workload within the interval of observation (which is [0, t), without loss of generality) arrives at time $t - C_j$ and executes over all C_j remaining time units of the interval (i.e., its full WCET).

Body: All preceding jobs fully contained in the interval, arriving exactly T_j units apart, also contribute to the workload, each with C_j time units.

Carry-in: Finally, the even earlier job (if any) executes as late as possible, within its busy period (i.e. C_j time units continuously until its deadline), in order for the time units of its execution contained within the interval [0, t) to be maximised.

It is this "jittery" effect (shifting the block of execution of the earliest interfering job of τ_j by $R_j - C_j$ time units to the right, compared to subsequent ones) that makes the carry-in case dominate the non-carry-in case. Obviously, there is some pessimism in this derivation because not every higher-priority task τ_j (if any at all) can have an execution pattern like this. Fortunately, Guan et al. [17] considerably reduced the pessimism by noting and proving that at most $m-1$ tasks can have carry-in workload (where m is the number of processors).

B. On the choice of priority assignment

Figure 8 shows a small-scale experiment with $m = 4$ processors and $n = 7$ tasks, for evaluating the choice of different priority assignments with respect to the success ratio reported by different schedulability tests (denoted by different colors). Deadline Monotonic (dotted line) does not perform very well. DkC with $k = 1.1$ (mixed line) barely underperforms DkC $\forall\ k \in \{0, 0.1, \ldots 2.0\}$ (dashed line), which in turn almost matches the optimal priority assignment (solid line), identified via top-down branch-and-bound enumeration.