Proc. of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017), Hsinchu, Taiwan on August 16-18, 2017.

A Scheduling Framework for Handling Integrated Modular Avionic Systems on Multicore Platforms

Alessandra Melani*, Renato Mancuso†, Marco Caccamo†, Giorgio Buttazzo*, Johannes Freitag‡, Sascha Uhrig‡
*Scuola Superiore Sant'Anna, Pisa, Italy, {alessandra.melani, g.buttazzo}@sssup.it
†University of Illinois at Urbana-Champaign, USA, {rmancus2, mcaccamo}@illinois.edu
‡Airbus Group Innovations, Munich, Germany, {johannes.freitag, sascha.uhrig}@airbus.com

***Abstract*—Although multicore chips are quickly replacing uniprocessor ones, safety-critical embedded systems are still developed using single processor architecture. The reasons mainly concern predictability and certification issues. This paper proposes a scheduling framework for handling Integrated Modular Avionics (IMA) on multicore platforms providing predictability as well as flexibility in managing dynamic load conditions and unexpected temporal misbehaviors of multicore. A new computational model is proposed to allow specifying a higher degree of flexibility and minimum performance requirements. Schedulability analysis is derived for providing off-line guarantees of real-time constraints in worst-case scenarios, and an efficient reclaiming mechanism is proposed to improve the average-case performance. Simulation and experimental results are reported to validate the proposed approach.**

I. Introduction

Multicore platforms are rapidly replacing uniprocessor counterparts in several application domains, since they allow increasing performance with a contained energy consumption, which also reflects on cooling, weight and space constraints. In spite of that, safety-critical embedded systems are still deployed on single-core computers. The main reasons that are delaying the adoption of multicore systems by avionic and automotive industries can be grouped in four categories: 1) temporal predictability; 2) migration of legacy software; 3) complexity of hardware architecture; and 4) software/hardware certification.

Currently, the Federal Aviation Administration (FAA) and other international certification authorities have serious concerns about the use of multicore platforms in Integrated Modular Avionics (IMA) due to various inter-core interference channels. A recent FAA's position paper, CAST-32 [1], establishes some guidelines to execute safety-critical applications on multicore. However, it partially enumerates the possible interfering channels on multicore, it is limited to two cores, and it does not address partitioning between applications in different subsystems, as mandated by the ARINC 653 avionic standard [2]. Existing uniprocessor implementations of IMA consist of a static cyclic executive that partitions applications into predefined time slots. In particular, ARINC 653 defines a schedule that is repeated every MAjor Frame (MAF). The MAF may be further divided into a sequence of MInor Frames (MIFs), which specify a fixed execution duration for each partition. According to this specification, each partition may be activated one or more times during the MAF.

In this work, we introduce a multicore IMA hierarchical scheduler that employs partitioned and Fixed Priority (FP) scheduling with periodic partition servers [3]. Compared to the traditional cyclic executive approach, we argue that the proposed scheduling framework preserves temporal isolation between partitions while providing higher flexibility in handling unexpected temporal misbehaviors[1] of a multicore platform. In fact, after known inter-core interference channels are mitigated and accounted for, residual and unmodeled architectural complexities can be further mitigated by allocating extra budget to critical partitions to cope with overruns for additional safety. Besides increasing schedulability over the classic cyclic executive scheme, the proposed approach has the advantage of managing unpredictable load variations with higher flexibility. Our efficient resource reclaiming mechanism exploits the unused extra budget allocated to more critical partitions in favor of less critical ones, in order to increase the average system performance.

Another important advantage of periodic servers versus cyclic scheduling is the possibility of handling aperiodic requests generated by asynchronous events and managing load variations through proper budget reclaiming mechanisms (e.g., [5]–[8]). This feature is particularly important in the presence of safety-critical partitions, where resources are overallocated to cope with worst-case scenarios and unpredictable overload conditions. Considering that such situations are typically very rare, a worst-case design without any reclaiming capability would lead to a very poor average-case performance. In a classical cyclic scheme, idle times coming from computations that terminate before their worst case are not exploited for executing other partitions, hence they are simply wasted, or at the best can be used to execute background activities.

Proposed approach. The proposed scheduling framework is specifically designed to provide functional safety and temporal isolation of IMA partions in spite of unmodeled complexities of a multicore architecture, as well as flexibility in handling load variations and asynchronous events. Its main features can be summarized as follows:

- The scheduler supports temporal and spatial partitioning between multiple applications. Temporal isolation is further strengthened by enforcing synchronous partition switching on all the cores (i.e., only one partition is active and running in parallel at any time). Hence, unforeseen temporal coupling among partitions is avoided by construction. However, within a partition, inter-core interference can be mitigated by adopting techniques like Single Core Equivalence (SCE) [9].
- Synchronous partition switching on all the cores provides enhanced temporal modularity on multicore; however, the resulting context switch overhead can be severe. The proposed scheme supports a limited preemption model for partitions to mitigate their switching overhead.
- The framework supports job skipping to allow less critical partitions to be designed with less stringent worst-case performance requirements. The skipping parameter associated to a task quantifies the reduction of service that can be tolerated for the corresponding functionality.
- To allow flexibility in specifying performance requirements, task execution follows an imprecise computation model, according to which any task instance first executes a manda-

[1]For instance, a recent research article [4] has uncovered a previously overlooked problem in the behavior of non-blocking caches that can invalidate the benefits of cache partitioning approaches.

978-1-5386-1898-1/17/$31.00 © 2017 IEEE

tory part, and then an optional part that integrates the former with additional operations. For example, in a flight control application, once structural integrity of the aircraft has been assured, additional calculations for fuel saving can be optionally performed.
- After all known inter-core interference channels are mitigated, unforeseen and unmodeled architectural complexities are mitigated by over-designing critical partitions with an extra budget for enhanced safety. Off-line schedulability analysis is carried out considering the overrun budget for the most critical partitions and assuming a worst-case job skipping scenario for the less critical ones.
- To compensate for the pessimism assumed in off-line analysis, an efficient on-line reclaiming mechanism is provided to possibly distribute the budget unused by a critical partition to less critical ones, according to a user-specified policy.

Finally, it is worth noticing that, in the special case where all partitions have the same period, the proposed server-based scheduling scheme is backward compatible with a cyclic executive scheme in which each partition executes for a single slot within the major cycle (equal to the server period).

Novel contributions. This paper has four main contributions. First, a general framework is presented to support next-generation IMA scheduling on multicore platforms, with the salient feature of limiting by construction the inter-core interference between logically independent tasks. Second, a new task model is proposed to allow the application designer to specify minimum functionality requirements in a more flexible way; the model generalizes two existing task models specifying performance degradation under overload conditions. Third, a schedulability analysis is derived for providing off-line real-time guarantees in worst-case scenarios, taking all the proposed system features into account. Finally, an efficient reclaiming mechanism is presented to compensate for pessimistic design choices and enhance the average-case performance. It allows exploiting the budget unused by a partition to reintegrate skipped jobs in other partitions according to a predefined policy.

Organization of the paper. The remainder of this paper is organized as follows. Section II reviews the related work. Section III presents the adopted system model and assumptions. Section IV describes the reclaiming algorithm. Section V describes the schedulability analysis. Section VI presents some experimental results carried out to evaluate the proposed approach. Finally, Section VII states the conclusions and outlines future work.

II. Related Work

The problem of inter-core interference in multicore architectures is well known and has been largely studied in the literature. Mancuso et al. [9], [10] proposed the Single Core Equivalence (SCE) technology for fixed priority scheduling. Under SCE, access to shared memory resources is strictly regulated using a set of OS-level techniques. For software certification, SCE allows m cores to be treated as independent processors of a conventional single-core chip, meaning that local workload changes require local re-validation only. However, SCE does not discuss how to extend IMA architectures in the presence of multicore platforms. Some approaches have been proposed to improve schedulability in single-core integrated modular avionics. Fohler [11], [12] proposed a slot shifting technique to reclaim spare capacities in a static schedule and incorporate aperiodic tasks and adaptive fault tolerance. Slot shifting was originally developed in the context of distributed real-time systems, but in principle could be adapted to IMA to cope with overruns. Agrawal et al. [13] presented a preliminary work for enabling slot-level time-triggered scheduling on COTS multicore platforms. This work addresses the problem of inter-core interference and provides a mitigation strategy for static off-line schedulers like cyclic executive.

This work shares a number of similarities with the Mixed-Criticality on Multi-core (MC^2) framework. The MC^2 framework [14]–[16] has been developed to execute mixed-criticality workloads on commercial multi-core systems, allowing different scheduling techniques to be adopted at different criticality levels. Similarly to MC^2, our work specifically addresses Integrated Modular Avionics and its Development Assurance Level (DAL) assignment[2], and introduces slack reclamation as a way to solve the problem of under-utilization of cyclic executive schemes. However, differently from MC^2, a key requirement in this work is to satisfy a strict extension to multi-core systems of the certification guidelines in DO-178B [17] and DO-178C [18]. In this extension, different cores are not allowed to execute in parallel applications belonging to different partitions. As such, a key feature of our is the enforcement of **synchronous partition switching** on all the cores. Additionally, unlike MC^2, we do not consider soft real-time constraints for low-criticality tasks: minimum functionality guarantees are always provided with *hard real-time constraints* at any assurance level, while efficiently reclaiming spare computational capacity. Resource reclaiming was also discussed in the context of MC^2 at a task-level, where reclaimed resources are assigned following a descending priority order. Conversely, (i) we investigate resource reclaiming at a partition level, and (ii) we study a number of different heuristics to select the partition that should receive the spare capacity, as described in Section IV-B.

Kim et al. [19] proposed a budgeted generalized rate-monotonic scheduling policy for uniprocessor IMA. The goal is to improve schedulability by assuming that all tasks are globally scheduled in priority order using a single-level scheduler. To support temporal modularity, each application is constrained to run within a total CPU utilization budget. The same authors presented a novel utilization bound [20] for a set of tasks in an IMA partition. The bound is useful at early development stage when task periods are known while task execution times are still unavailable. Other works [21], [22] proposed an extension of IMA to multicore platforms where the static cyclic executive partitioning is preserved. An optimized scheduling algorithm is proposed to handle I/O device management partitions as exclusive regions; hence, synchronization between device management partitions is achieved across cores. By presenting a constraint programming (CP) formulation, the authors find the minimum number of cores needed to schedule a given set of partitions on the multicore. Since the CP approach scales poorly, the same authors proposed a heuristic algorithm [21] that outperforms the CP approach in scalability. Additionally, the I/O partitions are consolidated in a dedicated core to simplify their management. In the proposed work, the problem of I/O partition synchronization is intrinsically solved since only one partition is active at any time. For instance, I/O partitions can be consolidated in a dedicated core, if needed.

The idea of bounding the effect of resource sharing by allowing tasks of the same class to run exclusively on the platform has been first formalized by the *Isolation Scheduling* model [23]. Optimality results were provided for this framework; moreover, its applicability in the context of mixed-criticality scheduling [24]

[2]Under DO-178B [17] and DO-178C [18], DAL is specified according to the severity of failure conditions (e.g., catastrophic (A); hazardous (B); major (C); minor (D); no effect (E)).

was discussed. According to this model, tasks are assigned different WCET estimates corresponding to different criticality levels; hence, at runtime the execution of low-criticality tasks can be degraded to accommodate the resource demand from high-criticality tasks. Cyclic executive schemes implementing the Isolation Scheduling model in a mixed-criticality system have been designed by Burns et al. [25].

III. System Model

This section formalizes the system model and describes the main features of the framework proposed to extend IMA scheduling to a multi-core environment.

A system consists of a set of N IMA partitions $\Pi_1, \ldots, \Pi_N$, running on a platform comprising m identical cores. Each partition Π_i is assigned three parameters (Q_i, P_i, π_i), where Q_i represents the allocated budget (or capacity), P_i its period, and π_i a fixed priority. At any time instant, *only one* partition can execute on the platform and partition switching is *synchronously* performed on all the cores. Therefore, each partition Π_i is implemented by m identical periodic servers $S_i^1, \ldots, S_i^m$, each of which defined by the same budget Q_i and period P_i of the corresponding partition. For the generic server S_i^c executing on core c, q_i^c denotes the current remaining budget, which is discharged proportionally whenever the server is scheduled for execution. Note that such a discharging rule ensures a synchronous partition switching.

While this feature avoids inter-partition interference by construction, it implies a higher runtime overhead, caused by the simultaneous cache flushing on all the cores, which can increase bus traffic due to cache misses. In the proposed framework, such an overhead is controlled by *limiting preemptability*, assuming that a partition Π_j with higher priority can preempt the currently executing partition Π_i only σ time units after Π_i was scheduled.

Each IMA partition Π_i runs an application composed of n_i periodic or sporadic[3] tasks $\tau_{i,1}, \ldots, \tau_{i,n_i}$, each of which is defined by a tuple $(C_{i,j}, D_{i,j}, T_{i,j}, p_{i,j})$, where $C_{i,j}$ denotes its worst-case execution time (WCET), $D_{i,j}$ its relative deadline, $T_{i,j}$ its period or minimum inter-arrival time, and $p_{i,j}$ its priority. Relative deadlines are assumed to be less than or equal to periods ($D_{i,j} \leq T_{i,j}$). A task $\tau_{i,j}$ is referred to as a *bound* task if its period is a multiple of the server's period and arrival times are coincident with server activation times, otherwise it is referred to as an *unbound* task.

To increase flexibility in specifying performance requirements, task execution follows an *imprecise computation* model [26]–[28], according to which any task instance (job) first executes a *mandatory* part, whose WCET is denoted by $M_{i,j}$, and then an *optional* part, whose WCET is denoted by $O_{i,j}$, where $C_{i,j} = M_{i,j} + O_{i,j}$. While the mandatory part of every job is always executed, its optional part may be fully *skipped* according to a skip parameter $S_{i,j}$, with the interpretation that an optional part can be skipped once every $S_{i,j}$ jobs, so that a minimal functionality requirement is guaranteed for the associated task [29], [30]. In the classical skip model, the skip parameter can range between 2 and infinity. In this paper, considering the existence of the mandatory part, the skip parameter can also be equal to 1, meaning that only the mandatory part is executed.

Job skipping can be exploited at design time to reduce the load in less critical partitions in favor of more critical ones, which can then be assigned a higher budget to tolerate transient overruns. Such a model encompasses the case of critical tasks (with no skipping capabilities) as a special case where $O_{i,j} = 0$. Analogously, a task with *soft* requirements and entirely skippable jobs can be modeled by setting $M_{i,j} = 0$. The proposed task model nicely integrates job skipping with imprecise computation, thus increasing the flexibility for specifying timing, as well as criticality constraints.

In the proposed framework, jobs selected to be skipped (at design time) can be reintegrated (fully or partially) at runtime by exploiting the spare budget saved by other high-priority partitions through the reclaiming mechanism explained in Section IV. Note that, since the proposed mechanism allows optional parts to be partially executed, in principle this could leave the system in an inconsistent state, for example if the budget is exhausted while a task is executing a critical section. However, a simple solution for avoiding inconsistency is to perform a budget check at the entrance of a critical section, and allow the selected job to continue only if the budget is enough to fully execute it. An alternative solution is to divide optional parts into chunks and donate budget only if it is sufficient to execute one or more chunks. Although the reclaiming method can be configured to readmit discrete portions of a skipped job, for the sake of simplicity, the rest of the paper assumes that any amount of a skipped job can be readmitted for execution, depending on the amount of spare budget saved by another partition. In summary, a task is defined by the following set of parameters:

$$(C_{i,j}, D_{i,j}, T_{i,j}, p_{i,j}, M_{i,j}, O_{i,j}, S_{i,j}).$$

A two-level hierarchical scheduler is assumed for managing application tasks. Partitions are scheduled according to a fixed priority (FP) algorithm and tasks within a partition are statically allocated to the m cores and managed by m corresponding periodic servers with the same parameters.

The set of tasks belonging to a partition and allocated to a core defines a *sub-application*. Tasks in a sub-application are executed by FP scheduling when the corresponding partition is scheduled and the corresponding server is running. Note that priorities of tasks belonging to different partitions are assumed to be unrelated. In the following, $hp(\Pi_i)$ and $lp(\Pi_i)$ denote the set of partitions with priority level greater than and lower than Π_i, respectively. Analogously, at the core level, $hp(\tau_{i,j})$ and $lp(\tau_{i,j})$ denote the set of tasks belonging to partition Π_i having priority level greater than or lower than $\tau_{i,j}$, respectively.

IV. Reclaiming Algorithm

This section presents the reclaiming algorithm adopted to redistribute the budget unused by partitions.

In order to describe how to reclaim the spare budget, two aspects need to be addressed: (i) How the spare budget can be transferred from one partition to another (*budget transfer mechanism*); (ii) How to select the partition(s) that receive the spare budget (*budget assignment policy*).

A. Budget transfer mechanism

To explain the proposed budget transfer mechanism, we first need to define the notion of *quiescent* sub-application inside the current server period.

Definition 1 (Quiescent sub-application)**.** *A sub-application of partition Π_i running on core c is defined to be* quiescent *in a given server period if all pending jobs have been completed (i.e., core c is currently idle) and no other jobs are going to be released before the end of the current server period.*

[3]The proposed results apply to both sporadic and periodic tasks. Periodic arrival patterns lead to more efficient budget reclaming. See Section IV.

Note that, in Definition 1, the completion time of a job corresponds to the completion of both its mandatory and optional parts, or its mandatory part only, according to the pattern imposed by the value of the skip parameter $S_{i,j}$.

In the proposed framework, the servers of a given partition are executed by FP scheduling and are forced to consume their budget (until exhaustion) when they are scheduled even though there is no pending workload to be executed. This constraint is needed to enforce the synchronicity among servers of the same partition and preserve system schedulability.

A first naive solution to reclaim the budget unused by some partition server in case of early completions would be to execute its own pending workload if any, thus avoiding partition switching. This type of reclaiming is denoted as *self-donation*. Note that such a pending workload would have been executed in the following server periods in case of no budget saving. The following example shows that self-donation could lead to a significant budget waste. Consider the task set in Table I, executing on a platform with $m = 2$ cores. In the example, partition Π_1 and Π_2 have parameters $(Q_1, P_1) = (5, 10)$ and $(Q_2, P_2) = (4, 12)$, respectively, and priorities assigned with Rate Monotonic (i.e., $\pi_1 > \pi_2$).

	Task	Core	$C_{i,j}$	$M_{i,j}$	$O_{i,j}$	$S_{i,j}$	$T_{i,j}$	$D_{i,j}$
Π_1	$\tau_{1,1}$	1	5	5	0	∞	10	10
	$\tau_{1,2}$	2	4	4	0	∞	10	10
	$\tau_{1,3}$	2	4	4	0	∞	40	40
Π_2	$\tau_{2,1}$	1	2	0	2	2	12	12
	$\tau_{2,2}$	1	6	6	0	∞	24	24
	$\tau_{2,3}$	2	4	4	0	∞	24	24
	$\tau_{2,4}$	2	4	4	0	∞	36	36

Table I: Task set parameters of the example in Fig. 1 and 2.

Figure 1 illustrates an example of self-donation, where partitions are allowed to reclaim their spare budget for executing their own pending workload. In the example, the first instances of $\tau_{1,1}$ and $\tau_{1,2}$ execute for their WCET, hence no budget is saved in the first server period. In the second period, both $\tau_{1,1}$ and $\tau_{1,2}$ complete earlier (at time 12), saving three units of budget on core 1 and two units on core 2. Note that self-donation allows $\tau_{1,3}$ to execute for additional 2 units of time, exploiting the saved budget and finishing earlier, at time 15. Such an earlier completion of $\tau_{1,3}$, however, does not lead to future donations. In fact, in the following server periods, $\tau_{1,1}$ executes for its WCET, thus the budget saved by $\tau_{1,2}$ cannot be donated to preserve a synchronous switching. Hence, the budget saved in partition Π_1 in the second server period is wasted.

Figure 2 shows what happens when self-donation is prohibited, so each sub-application can only consume the budget that would have been available in the worst case (e.g., just one time unit for $\tau_{1,3}$ in the second server instance). In this case, donation can take place at time 13, since two units of budget are simultaneously saved in both cores of partition Π_1 and can be donated to Π_2 to readmit the second job of $\tau_{2,1}$, which otherwise would have been skipped. On the second core, the donated budget is used to execute $\tau_{2,4}$, which was pending at the time of donation. Note, however, that inhibiting self-donation as in Figure 2 is not always the best way to reclaim the unused budget. In this example, in fact, $\tau_{1,3}$ could have continued executing from time 4 to 5, since the server on the first core was not ready to donate its budget at time 4. For these reasons, the proposed reclaiming strategy establishes a tradeoff between the two extreme cases described above in order to enforce the synchronicity among servers while reclaiming the budget.

The above example demonstrates that an effective reclaiming

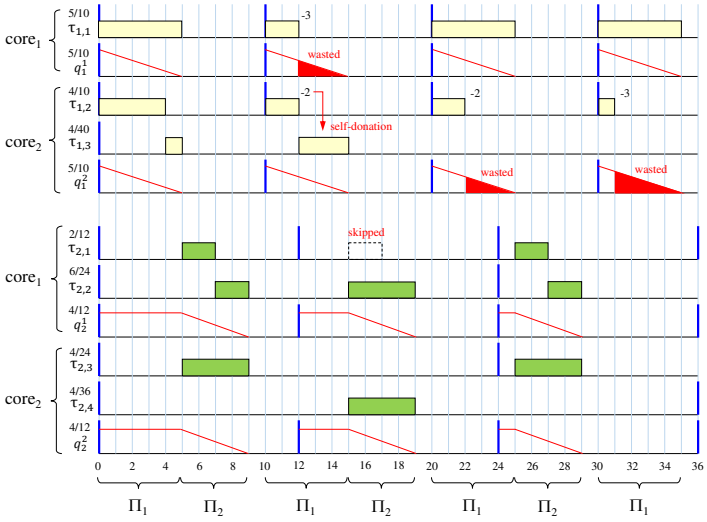


Figure 1: Example of budget waste arising from self-donation.

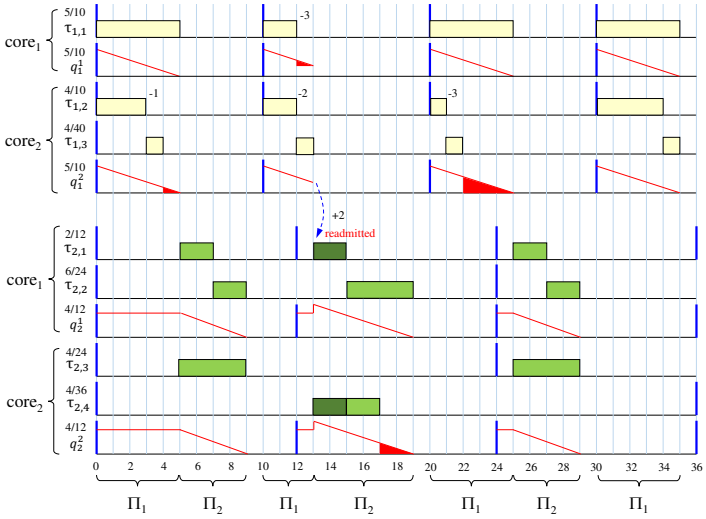


Figure 2: Example of readmitted job after budget donation.

strategy has to keep track of the saved budget. To achieve this goal, each server S_i^c also maintains a *worst-case budget*, denoted as $\overline{q}_i^c$, representing the budget that would have been consumed if all tasks were executing for their WCET. Such a server variable is updated according to the following rules:

- R1: At the beginning of each server period, $\overline{q}_i^c$ is set to Q_i;
- R2: When S_i^c is scheduled and some task is executing, $\overline{q}_i^c$ is decremented proportionally (with the same rate as q_i^c);
- R3: If a job completes earlier than its WCET, $\overline{q}_i^c$ is decremented by the saved amount $C_{i,j} - e_{i,j}$, being $e_{i,j}$ the actual job execution time, provided that $\overline{q}_i^c$ does not become negative (saturating to zero);
- R4: When S_i^c is scheduled and the core is idle, $\overline{q}_i^c$ is decremented proportionally (with the same rate as q_i^c) if the sub-application is quiescent or $\overline{q}_i^c = q_i^c$; else (if the sub-application is not quiescent and $\overline{q}_i^c < q_i^c$) it is kept constant.

The following example clarifies the rules for updating the worst-case budget. Consider a partition Π_1 with parameters $(Q_1, P_1) = (10, 20)$ executing the task set in Table II on a platform with $m = 2$ cores.

In Figure 3, solid-line rectangles represent actual task execution, while dashed-line rectangles denote additional execution up to the worst-case. On the first core, $\tau_{1,1}$ terminates its first job

Task	Core	$C_{i,j}$	$M_{i,j}$	$O_{i,j}$	$S_{i,j}$	$T_{i,j}$	$D_{i,j}$
$\tau_{1,1}$	1	5	5	0	∞	20	20
$\tau_{1,2}$	1	5	5	0	∞	40	40
$\tau_{1,3}$	1	5	5	0	∞	40	40
$\tau_{1,4}$	2	5	5	0	∞	20	20
$\tau_{1,5}$	2	4	4	0	∞	20	20

Table II: Task set parameters of the example in Fig. 3.

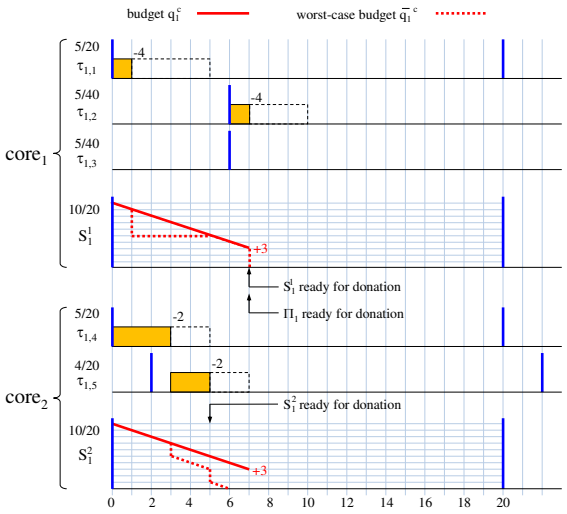


Figure 3: Example showing the rules for updating the worst-case budget and the budget donation mechanism.

at time $t = 1$, saving 4 units of time. Hence, $\overline{q}_1^1$ is decremented linearly while Π_1 is scheduled (rule R2), and is decremented by 4 units at time 1 (rule R3). Then, since the sub-application is not quiescent and $\overline{q}_1^1 < q_1^1$, $\overline{q}_1^1$ is kept constant until time $t = 5$ (Rule R4), and then it follows the same trend as q_1^1 in [5,7], since now $\overline{q}_1^1 = q_1^1$. At time $t = 7$, $\overline{q}_1^1$ is further decremented by 4 units (saturating to zero) due to the earlier completion of $\tau_{1,2}$. On the second core, the worst-case budget $\overline{q}_1^2$ is linearly decremented during task execution (rule R2), and then decreased by two units at times $t = 3$ and $t = 5$, since both tasks complete earlier by saving two units of time (rule R3). At time $t = 5$, the sub-application becomes quiescent, since there are no more pending jobs within the server period, so $\overline{q}_1^2$ is linearly decremented saturating to zero (rule R4). As explained later, once a sub-application becomes quiescent, it is available to donate its spare budget, so the value of the worst-case budget is only relevant when the sub-application is not quiescent.

The following lemma establishes two conditions that enable the donation mechanism to be triggered without violating the system schedulability.

Lemma 1. *A server becomes* ready for donation *when at least one of the following conditions is satisfied.*

Condition 1. *A server S_i^c is ready for donation if its current server budget is greater than zero ($q_i^c > 0$) and the corresponding sub-application is quiescent.*

Condition 2. *A server S_i^c is ready for donation if its current server budget is greater than zero ($q_i^c > 0$) and its worst-case budget is equal to zero ($\overline{q}_i^c = 0$).*

Proof. Condition 1 ensures that all pending jobs have been completed and no jobs will be released by the end of the current server period (by Definition 1), hence donation can occur without violating system schedulability. Condition 2 is also sufficient to perform donation safely, because it ensures that the amount of execution occurred in the current server period is the same as it would have been in the worst case. Indeed, by the budget update rules given above, $\overline{q}_i^c$ can only become lower than q_i^c due to earlier task completions. □

Then, the budget donation mechanism is triggered according to a Donation Rule that enforces the synchronous partition switching upon donation. This rule also requires that the amount of spare budget is at least σ time units (i.e., at least equal to the length of the non-preemptability interval). Being σ certainly greater than the switching overhead, this condition ensures that the partition receiving the budget will not execute more than expected. In fact, if a budget smaller than σ is donated to another partition, the switching overhead could lead to a violation of system schedulability.

Donation Rule: A partition Π_i becomes *ready for donation* as soon as all its servers S_i^c, $c = 1, \ldots, m$, are ready for donation, provided that the amount of spare budget is at least σ time units.

The Donation Rule ensures that, for all servers of partition Π_i, either Condition 1 or 2 is verified, hence donation can safely take place. If a certain server is ready for donation because Condition 2 is verified (its worst-case budget is exhausted), but the other servers are not ready for donation yet, if the current budget is still greater than zero, the server can exploit the time until the partition becomes ready for donation to execute its pending workload (if any). Since the budget donation mechanism requires synchronicity among all the cores, this choice appears to be reasonable since that capacity would be wasted anyway. In addition, this mechanism has the advantage of allowing a synchronous behavior on all the cores. Therefore, we allow a server that is ready for donation to consume its budget to execute its pending workload while waiting for the other servers to become ready for donation.

In the example of Figure 3 (which assumes $\sigma = 0$ for simplicity), server S_1^1 becomes ready for donation when its worst-case budget is exhausted, at time 7 (by Condition 2). Server S_1^2, instead, becomes ready for donation at time 5, when the corresponding sub-application becomes quiescent and thus Condition 1 is verified. In the example, the Donation Rule establishes that partition Π_1 is ready for donation at time 7, corresponding to the earliest time instant when both servers are ready for donation.

Once a donation is triggered by a given partition Π_i, the amount of saved budget can be exploited by another partition Π_j (selected as explained in Section IV-B) by incrementing the budget (and the worst-case budget) of all its servers by an amount equal to the capacity saved by Π_i at the time the donation was triggered.

It is worth observing that the proposed reclaiming mechanism becomes particularly efficient when tasks are strictly periodic. In fact, in case of sporadic arrivals, Condition 1 can only be verified if the specified minimum inter-arrival times are such that subsequent task arrivals do not fall within the current server period. In addition, the proposed approach achieves its full potential if tasks periods are multiples of the server period and arrival times are coincident with the server activation times (i.e., in the case of bound tasks). In fact, in this case there would be no intermediate arrivals falling inside the server period, hence Condition 1 could be simply triggered when all pending jobs have been completed. This scenario also avoids the fragmentation of budget saving due to intermediate arrivals, hence it has the advantage of favouring a synchronous budget saving on all the cores, so that the spare budget can be more likely reclaimed and redistributed to other partitions.

A final observation concerns the non-preemptability interval of σ time units enforced every time a partition is scheduled. As mentioned in Section III, this parameter has the purpose of controlling partition switching overhead. Remarkably, if the Donation Rule is verified for partition Π_i during the non-preemptability interval (i.e., before σ time units have elapsed since the last time Π_i was scheduled), it is convenient to trigger the budget transfer mechanism. Otherwise, if the donation mechanism is not triggered, the spare capacity of Π_i would be simply wasted.

B. Budget assignment policy

Once the budget donation mechanism is triggered, a budget assignment policy should select the partition receiving the spare budget, according to a system-level strategy. This section describes some heuristics that can be adopted to select the receiving partition.

First of all, note that, in order to preserve schedulability, a budget saved by partition Π_i can only be donated to a partition Π_j with lower priority ($\pi_j < \pi_i$). In fact, if the budget should be donated to a partitions Π_h with priority $\pi_h > \pi_i$, the schedulability of the system could be jeopardized, since a partition Π_k with intermediate priority $\pi_i < \pi_k < \pi_h$ could be delayed beyond its deadline by the extra execution of Π_h. This consideration has been originally formalized for the Priority Exchange Server [31] to handle aperiodic requests along with a set of hard periodic tasks. The only exception to this rule is given by the possibility of donating the budget saved by Π_i to the partition having a priority level immediately higher than π_i (if any), since there would be no intermediate partitions that could be delayed. Such a reasoning leads to the following constraint:

Constraint 1. *If partition priorities are ordered such that $\forall i < j$, $\pi_i > \pi_j$, than a budget saved by a partition Π_i can only be donated to a partition $\Pi_j \in lp(\Pi_i) \cup \{\Pi_{i-1}\}$.*

Ensuring Constraint 1 is vital to preserve the schedulability while reclaiming the budget. In order to explain the proposed budget assignment policy, it is worth defining the concept of eligible jobs and partitions.

Definition 2. *A job is defined to be* eligible *for donation if it is active and its optional part is selected for being skipped.*

Definition 3. *A partition Π_i is defined to be* eligible *for donation if at least one of its jobs is eligible for donation.*

When a donation is triggered, the amount of spare budget donated to an eligible partition (according to Constraint 1) can be used to execute one or more eligible jobs. As soon as a job becomes eligible, it is inserted in a queue of eligible jobs, from which it is removed upon its absolute deadline. Since tasks are statically allocated to the m cores, a queue of eligible jobs needs to be maintained for each core; jobs in each queue can be ordered by decreasing priorities. Another constraint is needed to account for the execution progress of eligible jobs.

Constraint 2. *If a certain amount of budget is donated at time t, a job can benefit of the donation only if its mandatory part has been completed at time t.*

Once a given partition is selected as the recipient of the budget donation, the spare budget is used, on each core, to execute the optional part of eligible jobs allocated to that core, following their priority ordering, until there are no more optional jobs to execute. The remaining budget (if any) can be consumed until exhaustion to execute other pending workload.

In the following, three possible heuristics are considered for selecting the eligible partition receiving the spare budget. Such heuristics should be applied only to eligible partitions satisfying Constraints 1 and 2, in order to account for the priority level and the execution progress of eligible jobs.

Priority-based approach. At each decision point, it selects the eligible partition with the highest-priority. This policy, however, does not consider the amount of workload that can be actually reclaimed.

Fairness-based approach. This approach aims at redistributing the spare budget in a fair way. To do so, it keeps track of the total amount $Q(t)$ of reclaimed budget in the interval $[0, t]$ and tries to minimize for each partition the lag between the ideally fair share of budget and the actual amount of budget granted to partition Π_i. Therefore, the budget is granted to the partition with eligible jobs that currently maximizes

$$\frac{m \cdot Q(t)}{N'} - o_i(t), \tag{1}$$

where $N' \leq N$ is the number of partitions encapsulating at least one skippable task, $m \cdot Q(t)/N'$ represents the ideally fair share of budget at time t, and $o_i(t)$ denotes the sum over all cores of the portions of the optional parts readmitted by the reclaiming mechanism in the interval $[0, t]$.

Max-workload approach. This approach selects the partition that could benefit more from the budget donation in terms of re-enabled workload, that is, the partition that would be able to execute more workload inside the execution interval granted by the donation.

The policies described above would become slightly more complex if the considered task model did not allow imprecise computation, because they would need to take into account the amount of spare budget to select tasks whose WCET fits the available budget. By allowing tasks to execute an arbitrary portion of their optional part, partitions can be selected to receive the spare budget independently of its actual amount.

C. Evaluation metrics

This section defines some metrics for evaluating and comparing the different policies proposed above.

In systems that support imprecise computation, a commonly used metric is the error ϵ_i in the result produced by a given task τ_i, defined as the length of the portion of the optional part discarded in the schedule. If O_i is the worst-case duration of the optional part and o_i is the actual processor time assigned to the optional part by the scheduler, the error of task τ_i is defined as $\epsilon_i = O_i - o_i$. The average error $\bar{\epsilon}$ on the task set is defined as

$$\bar{\epsilon} = \frac{1}{n} \sum_{i=1}^{n} w_i \cdot \epsilon_i, \tag{2}$$

where w_i is a weight indicating the relative importance of τ_i in the task set.

The metric $\bar{\epsilon}$ is not directly used in this paper as it does not apply to systems based on partitions. Additionally, to capture the effect of the proposed online budget reclaiming mechanism, a new metric is introduced that is a function of the considered time interval. Namely, the *system optional workload rate* $\rho(t)$ in the interval $[0, t]$ is defined as the average ratio between the amount

of optional parts of tasks in Π_i actually executed, referred to as $o_i(t)$, and the total amount $O_i(t)$ of optional execution activated in the considered interval:

$$\rho(t) = \frac{1}{N} \sum_{i=1}^{N} w_i \frac{o_i(t)}{O_i(t)}, \tag{3}$$

where the weight w_i refers to the relative importance of partition Π_i in the system. We assume that the variable $o_i(t)$ is updated by the operating system.

In order to evaluate the fairness in the allocation of the spare budget, a *load balance index* $\beta(t)$ is defined as the average squared error between the ideally fair assignment $m \cdot Q(t)/N'$ and the budget $o_i(t)$ effectively assigned to partitions that contains at least one skippable task:

$$\beta(t) = \frac{1}{N'} \sum_{\substack{i \in [1,N] \\ \exists j \mid S_{i,j} \neq \infty}} \left(\frac{m \cdot Q(t)}{N'} - o_i(t) \right)^2. \tag{4}$$

The different reclaiming strategies presented above are evaluated in Section VI according to these metrics by a simulation-based approach.

V. Schedulability Analysis

This section presents an off-line schedulability test for an IMA system compliant with the proposed model. For each partition, the worst-case scenario is constructed by assuming the execution pattern imposed by the associated task skipping parameters, where, for each task $\tau_{i,j}$ having $S_{i,j} > 1$, its first $S_{i,j} - 1$ jobs are assumed to be fully executed. Since tasks are statically partitioned to cores, the analysis is restricted to a single-core scenario. System schedulability is then guaranteed by verifying the schedulability individually on each core. The analysis will be first presented considering a fully-preemptive system with $\sigma = 0$, and then extended to consider a limited-preemptive scenario where $\sigma > 0$.

Background: Davis and Burns [3] described an exact uniprocessor schedulability test for the Periodic Server (PS) [32], which assumes a hierarchical framework where fixed-priority preemptive scheduling is adopted both at global and local levels, and servers are invoked every P_i time units. At each server execution period, the capacity is linearly discharged even though there are no ready tasks to execute.

In this setting, a schedulability analysis is carried out considering a critical instant for any task $\tau_{i,j}$ scheduled under a given server [3]. In this scenario, the total workload generated by $\tau_{i,j}$ and all higher-priority tasks in a time window of length L is given by:

$$W_{i,j}(L) = C_{i,j} + \sum_{\tau_{i,k} \in hp(\tau_{i,j})} \left\lceil \frac{L + J_{i,k}}{T_{i,k}} \right\rceil C_{i,k}, \tag{5}$$

where $J_{i,k}$ is the task jitter with respect to the server release. It holds that $J_{i,k} = 0$ if $\tau_{i,k}$ is a bound task, and $J_{i,k} = (P_i - Q_i)$ if $\tau_{i,k}$ is unbound. The length of the gaps during server periods (excluding the last one) is

$$G_{i,j}(L) = \left(\left\lceil \frac{W_{i,j}(L)}{Q_i} \right\rceil - 1 \right) (P_i - Q_i). \tag{6}$$

The extent to which the busy period L extends into the final server period is

$$Z_{i,j}(L) = L - \left(\left\lceil \frac{W_{i,j}(L)}{Q_i} \right\rceil - 1 \right) P_i. \tag{7}$$

Finally, the total response time of a task $\tau_{i,j}$ is given by the fixed-point iteration of the following formula, starting with $L^{(0)} = C_{i,j} + \left(\left\lceil \frac{C_{i,j}}{Q_i} \right\rceil - 1 \right) (P_i - Q_i)$:

$$L^{(x+1)} = W_{i,j}(L^{(x)}) + G_{i,j}(L^{(x)}) + \sum_{\Pi_k \in hp(\Pi_i)} \left\lceil \frac{(Z_{i,j}(L^{(x)}))_0}{P_k} \right\rceil Q_k, \tag{8}$$

where the notation $(a)_0$ stands for $\max(0, a)$.

Figure 4 illustrates the critical instant scenario for a task $\tau_{i,j}$ scheduled under a given server. Assuming that $\tau_{i,j}$ is an unbound task, in the worst-case it is released simultaneously with all its higher-priority tasks right after the corresponding server has exhausted its budget Q_i (dashed red line in Figure 4). Hence its jitter $J_{i,j}$ with respect to the server release is equal to $P_i - Q_i$. The figure also shows the outstanding portion $Z_{i,j}$ of the busy period extending into the final server period. If $\tau_{i,j}$ is a bound task, instead, the critical instant scenario corresponds to the case in which $\tau_{i,j}$ and all its higher-priority tasks are released simultaneously with the server activation, hence the task jitter is simply equal to zero.

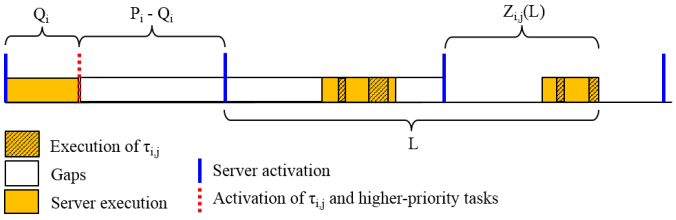


Figure 4: Critical instant scenario for an unbound task scheduled under a periodic server.

Fully-preemptive scheduling: The schedulability analysis presented in [3] can be easily adapted to our case if a fully-preemptive scheduler (i.e., with $\sigma = 0$) is assumed. Specifically, since we allow the optional part of each task to be skipped according to the value of $S_{i,j}$, while mandatory parts of all task instances are never skipped, the total workload $W_{i,j}(L)$ at priority level j or higher, generated by tasks belonging to $\tau_{i,j}$'s partition and allocated to the same core, is given by:

$$\begin{aligned} W_{i,j}(L) = C_{i,j} &+ \sum_{\substack{\tau_{i,k} \in hp(\tau_{i,j}) \\ \cap\ \text{CORE}(\tau_{i,j})}} \left\lceil \frac{L + J_{i,k}}{T_{i,k}} \right\rceil M_{i,k} + \\ &\sum_{\substack{\tau_{i,k} \in hp(\tau_{i,j}) \\ \cap\ \text{CORE}(\tau_{i,j})}} \left(\left\lceil \frac{L + J_{i,k}}{T_{i,k}} \right\rceil - \left\lfloor \left\lceil \frac{L + J_{i,k}}{T_{i,k}} \right\rceil \frac{1}{S_{i,k}} \right\rfloor \right) O_{i,k}, \end{aligned} \tag{9}$$

where $\text{CORE}(\tau_{i,j})$ denotes the set of tasks allocated to the same core as $\tau_{i,j}$. Therefore, the analysis in [3] needs to be modified by simply replacing Equation (5) with Equation (9). Then, schedulability can be checked by performing fixed-point iteration of Equation (8), with the summation being restricted to all tasks belonging to $hp(\tau_{i,j}) \cap \text{CORE}(\tau_{i,j})$.

Limited-preemptive scheduling: If a non-preemptive interval of $\sigma > 0$ time units is enforced every time a partition is scheduled, the analysis in [3] needs to be further adapted. In particular, the contribution of σ does not need to be explicitly accounted for in complete server periods, as the only information incorporated by the schedulability test is the amount of budget Q_i that needs to be granted to partition Π_i during any complete server period P_i.

The non-preemptive interval comes into play when considering the execution of the last piece, whose length is given by:

$$Y_{i,j}(L) = W_{i,j}(L) - \left(\left\lceil \frac{W_{i,j}(L)}{Q_i} \right\rceil - 1 \right) Q_i. \quad (10)$$

For this interval, we need to consider (i) the potential blocking from lower-priority servers; and (ii) the interference from higher-priority servers. While the former contribution is simply bounded by σ, the interference from higher-priority servers can be computed by observing that, in the case of a Periodic Server, it is maximized when all of them are released synchronously with the start time of the last server instance of Π_i. To account for the length of the non-preemptive interval for partition Π_i, two cases have to be distinguished:

Case 1). $Y_{i,j}(L) \leq \sigma$. In this case, after suffering the interference imposed by higher-priority servers, the last chunk executes non-preemptively (see Figure 5a)). The start time of the last piece (relative to the start time of the last server period), can be computed taking into account the *self-pushing phenomenon* that arises in non-preemptive scheduling, which requires to carry out the response time analysis for all job instances falling inside the so called *Level-i Active Period* [33]. Hence, the start time of the last chunk must be computed by the response time analysis for non-preemptive scheduling as described in [33], [34], assuming a worst-case blocking time of σ time units from lower-priority servers and a worst-case execution time of $Y_{i,j}(L)$. Since the value of $Y_{i,j}(L)$ is dependent on the current length of the busy period, this computation requires to carry out a nested fixed-point iteration. Once the start time of the last chunk has been computed, the current values of $W_{i,j}(L)$ and $G_{i,j}(L)$ can be added up to get a new response time value. The procedure has to be repeated until unschedulability is detected ($L^{(x+1)} > D_{i,j}$) or convergence is reached ($L^{(x)} = L^{(x+1)}$), determining a positive schedulability result.

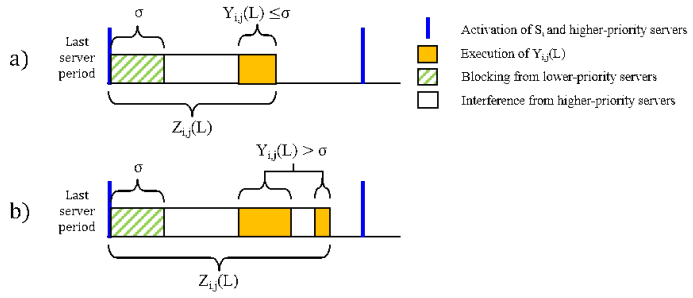


Figure 5: Impact of limited preemption in the last server period.

Case 2). $Y_{i,j}(L) > \sigma$. In this case, one non-preemptive chunk is not sufficient to execute $Y_{i,j}(L)$ entirely. Therefore, a server Π_i could be scheduled and de-scheduled multiple times before being able to finalize $Y_{i,j}(L)$. Each time it is scheduled, it will execute non-preemptively for σ time units, and then will be possibly preempted by higher-priority servers. This scenario is illustrated in Figure 5b). In this case, a tight schedulability analysis should be able to detect off-line whether the final part of the last chunk will execute preemptively or not, which in turns depends on the interleaving among higher-priority servers. Since in the worst-case it is not possible to take advantage of a last non-preemptive chunk, the considered problem can be analyzed by analogy with the Limited Preemptive scheduling model with Deferred Preemptions [34]. Specifically, an upper-bound on the response time of $\tau_{i,j}$ can be simply computed by the fixed-point iteration of the following formula, starting with $L^{(0)} = C_{i,j} + \left(\left\lceil \frac{C_{i,j}}{Q_i} \right\rceil - 1 \right) (P_i - Q_i)$:

$$L^{(x+1)} = W_{i,j}(L^{(x)}) + G_{i,j}(L^{(x)}) + \sigma + \sum_{\substack{\tau_{i,k} \in hp(\tau_{i,j}) \\ \cap \text{ CORE}(\tau_{i,j})}} \left\lceil \frac{(Z_{i,j}(L^{(x)}))_0}{P_k} \right\rceil Q_k. \quad (11)$$

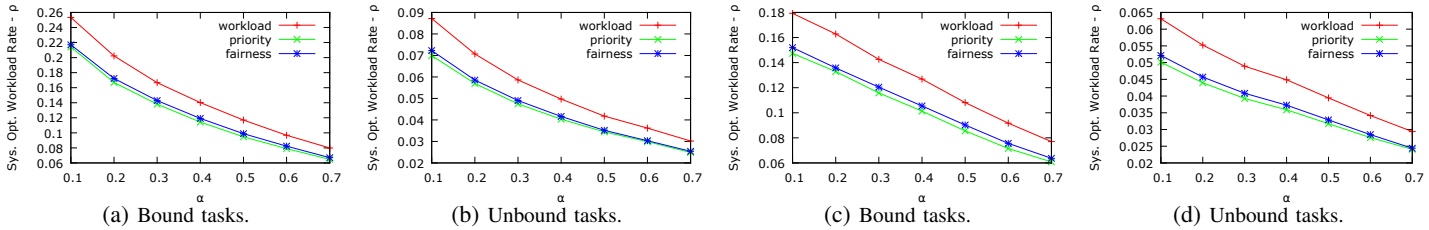
Note that Equation (11) represents a valid upper-bound on the response time of $\tau_{i,j}$ also in Case 1), thus avoiding the nested iteration at the cost of adding some pessimism in the analysis. The benefit of using Equation (11), however, is that the analysis becomes much simpler and can be limited to the first job of each task. A block diagram that provides the big pictue on how the proposed schedulability analsyis can be used is provided in [35].

VI. Experimental Evaluation

The performance of the proposed scheduling framework has been evaluated by extending SimSo [36], an open-source simulator[4] written in Python that targets real-time multiprocessor architectures. The three budget assignment heuristics proposed in Section IV-B have been evaluated on a synthetic workload, randomly generated as described in the following subsection. Additional experiments are available in [35].

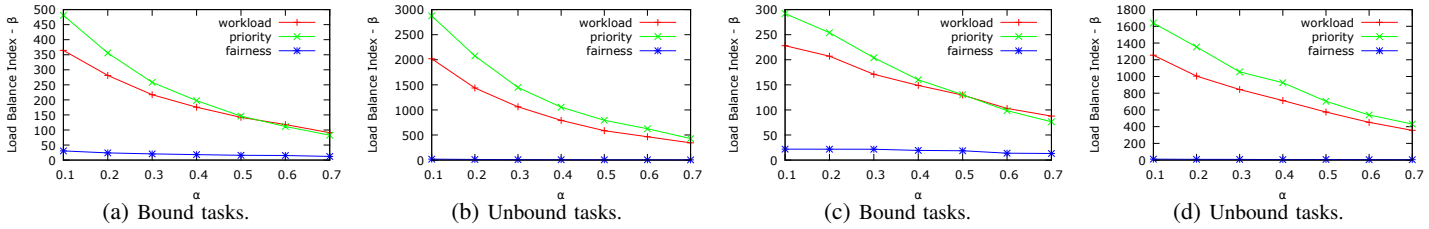
Workload generation: The proposed experiments consider a system with 10 IMA partitions. Different configurations are tested with a number of cores m that goes from 2, in line with [1], to 8, which represents the typical upper-bound in multi-core systems for avionic applications. Partition servers are assigned harmonic periods, randomly chosen in the set $\{25, 50, 100, 200, 400\}$. Server utilizations are generated by UUniFast [37], and server budgets are computed accordingly. Partitions are assigned priorities according to Rate Monotonic (RM). For each partition, 3 to 5 tasks are generated, in line with the typical number of tasks in avionic applications [38]. In the case of bound tasks, periods are selected randomly in the following set of values: $\{25, 50, 100, 200, 400, 800, 1600, 2400\}$; additionally, for each task $\tau_{i,j}$, $T_{i,j} \geq P_i$. In the case of unbound tasks, periods are selected with uniform distribution in the interval $[P_i, 2400]$. In both cases, relative deadlines are set equal to periods. Then, for each task, the utilization of its mandatory part is selected with uniform distribution in the interval $[0.01, 0.2]$. The total utilization of mandatory parts does not exceed $0.8 \cdot \frac{mQ_i}{P_i}$. In other words, the maximum utilization of mandatory parts in a partition is not higher than 80% of the budget assigned to the partition. This choice allows evaluating the system in a condition of heavy load while having some slack for the execution of optional workload. Then, the value of $M_{i,j}$ is computed accordingly. Additionally, the first five partitions (in priority order) are assumed to be highly-critical, in the sense that they are designed with no skipping capabilities ($S_{i,j} = \infty$). The remaining five partitions have an optional part whose utilization is uniformly selected in the interval $[0.01, 0.2]$, provided that the total utilization of optional parts in each partition does not exceed $\frac{mQ_i}{2P_i}$. Indeed, if the additional optional workload is excessive, it will always be skipped, thus providing no insight on the efficiency of the proposed reclaiming strategies. Conversely, a low amount of optional workload would make reclaiming strategies indistinguishable. In this context, we set the maximum utilization of optional workload to be about 50% of the partition budget. In this way, optional workload can bring non-critical partitions to be loaded at roughly 130% of their budget when mandatory parts run for their worst-case length $M_{i,j}$. Once period and utilization have been selected, the length of each optional part $O_{i,j}$ is computed accordingly. All tasks

[4]Code available at: https://github.com/edwardjamming/simso-MC-IMA



(a) Bound tasks. (b) Unbound tasks. (c) Bound tasks. (d) Unbound tasks.

Figure 6: System optional workload rate in the case of bound (a) and unbound (b) tasks with log-uniform runtimes; and in case of bound (c) and unbound (d) tasks with uniform runtimes.



(a) Bound tasks. (b) Unbound tasks. (c) Bound tasks. (d) Unbound tasks.

Figure 7: Load balance index in the case of bound (a) and unbound (b) tasks with log-uniform runtimes; and in case of bound (c) and unbound (d) tasks with uniform runtimes.

belonging to non-critical partitions have a skipping parameter $S_{i,j} = 1$ (i.e., all optional parts are skipped by default). Tasks are then statically partitioned to the available cores by a best-fit bin-packing heuristic that allocates each task to the least utilized core. Task priorities are set according to RM. Simulations have been carried out with $\sigma = 0$, as this parameter mostly affects the scheduling of partitions but does not significantly impact the amount of budget that can be reclaimed at runtime. Finally, for all the partitions, the weight w_i is set equal to 1 (see Equation (3)).

Only system configurations deemed schedulable by the test in Section V have been considered in the evaluation. In each set of experiments, 500 schedulable task-sets have been generated for each value on the x-axis. The duration of each simulation experiment is set equal to the task set hyperperiod, assuming that tasks are synchronously activated at the beginning of the execution.

Simulation results: In a first set of experiments, the system optional workload rate (as defined in Section IV-C) has been measured as a function of the amount of execution allowed to each partition. In particular, the actual execution time of each task is generated as a random integer selected with log-uniform distribution in the interval $[\alpha \cdot M_{i,j}, M_{i,j}]$, with $\alpha \in [0.1, 0.7]$, and the performance of the different budget assignment policies proposed in Section IV-B is evaluated as a function of α. The results depicted in Figures 6a) and 6b) correspond to the bound and unbound case, respectively. As expected, the workload-based policy dominates the other two approaches in terms of readmitted optional workload for all values of α. In addition, the system optional workload rate in the case of bound tasks is about 2.8 times higher compared to the case of unbound tasks, confirming that the proposed reclaiming algorithm has better performance with bound tasks. The observed trend confirms that the proposed workload-based reclaiming strategy is effective to maximize the overall execution of optional workload. However, the performance of this strategy is not significantly better than the other approaches. This is because the lengths of the optional parts for all tasks are generated following the same law. Hence, good performance is achieved even if the selection is not purely based on the amount of pending optional workload.

Additionally, even when $\alpha = 0.7$ (i.e., all tasks execute close to their worst case), some budget reclaiming may still occur (8% and 3%, respectively), since it is unlikely that all the tasks fully utilize their partition budgets. For example, if all sub-applications inside a partition become quiescent before depleting the budget, the reclaiming mechanism is triggered according to Condition 1 (see Section IV-A).

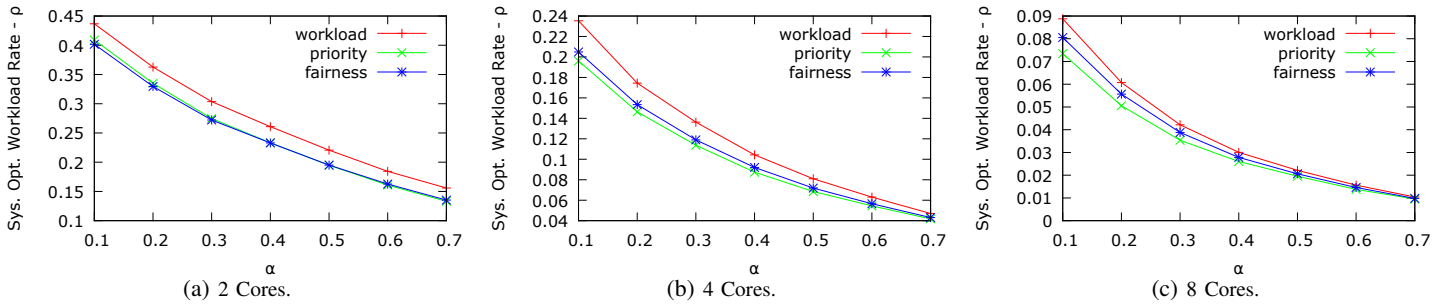
A second set of experiments has been carried out to evaluate the load balance index considering the value of globally reclaimed budget at the end of the simulation. The results reported in Figures 7a) and 7b) demonstrate that the fairness-based approach is effective to evenly distribute the available budget among different non-critical applications. This is important to guarantee that no software component is starved even if sufficient budget is reclaimed to guarantee its execution.

The system optional workload rate is further evaluated as the number of cores is increased from 2 to a maximum of 8. For this case, we consider a lower utilization for the optional workload. Specifically, the total utilization of the optional parts in each partition is $\frac{mQ_i}{4P_i}$ (i.e., 25% of the partition budget), instead of $\frac{mQ_i}{2P_i}$ as in the previous experiments. As discussed, an optional utilization of 50% already brings the system close to saturation for large values of α. In this experiment, a lower utilization value is selected to better appreciate the variation in terms of optional workload rate as the number of cores is increased.

Next, the system optional workload rate in case of bound task with log-uniform runtime distribution is evaluated considering a different number of cores in the system. Figure 8 reports the results for system setups with $m = 2, 4$ and 8 cores, respectively. In this set of experiments, the number of tasks per partition is calculated as $2.5m$. The plots in Figure 8 suggest that the amount of optional workload executed is inversely proportional to the number of cores in the system. This behavior can be explained considering two main aspects: (i) as the number of cores increases, a larger amount of optional workload is generated; and (ii) despite the larger amount of workload, there is marginally less opportunity for donation since the budget saving has to be realized simultaneously on all the cores to trigger reclaiming.

VII. Conclusions

This paper proposed a scheduling framework for handling Integrated Modular Avionics (IMA) on multicore platforms pro-



(a) 2 Cores. (b) 4 Cores. (c) 8 Cores.

Figure 8: System optional workload rate with bound tasks and variable number of cores.

viding predictability as well as flexibility in managing unexpected temporal misbehaviors of multicore. To eliminate the inter-core interference across different partitions, servers executing in parallel within the same partition have the same budget and period for all the cores; hence, partitions are synchronously switched on the platform. To safely handle unexpected overload conditions without sacrificing the average-case performance, applications are first guaranteed off-line by allocating resources according to pessimistic worst-case scenarios. However, at runtime an efficient online resource reclaiming mechanism exploits early completions and redistributes the unused budget according to a predefined donation strategy. Three different donation heuristics have been proposed and evaluated by using simulations. With respect to the classical cyclic executive approach, the achieved results demonstrated that the proposed framework is a promising solution for implementing safety-critical IMA systems on multicore platforms.

ACKNOWLEDGMENT

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1302563 and CNS-1646383. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF and other sponsors.

REFERENCES

[1] FAA position paper on multicore processors, CAST-32 (Rev 0), May 2014. [Online]. Available: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf

[2] Airlines Electronic Engineering Committee (AEEC) and Aeronautical Radio Inc., Avionics Application Standard Software Interface - ARINC Specification 653, January 1997. [Online]. Available: https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=1487

[3] R. Davis and A. Burns, "Hierarchical fixed priority preemptive scheduling," in *RTSS*, 2015.

[4] P. Kumar Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *RTAS*, 2016.

[5] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *RTSS*, 2000.

[6] G. Bernat and A. Burns, "Multiple servers and capacity sharing for implementing flexible scheduling," *Real-Time Systems*, vol. 22, no. 1, pp. 49–75, 2002.

[7] C. Lin and S. Brandt, "Improving soft real-time performance through better slack reclaiming," in *RTSS*, 2005.

[8] R. Pellizzoni and M. Caccamo, "M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms," *Real-Time Systems*, vol. 40, no. 1, 2008.

[9] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "WCET(m) estimation in multi-core systems using Single Core Equivalence," in *27th Euromicro Conference on Real-Time Systems*, July 2015, pp. 174–183.

[10] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo, "WCET derivation under Single Core Equivalence with explicit memory budget assignment," in *29th Euromicro Conference on Real-Time Systems*, June 2017.

[11] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *RTSS*, 1995.

[12] ——, "Adaptive fault-tolerance with statically scheduled real-time systems," in *EMWRTS*, 1997.

[13] A. Agrawal, G. Fohler, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Slot-level time-triggered scheduling on COTS multicore platform with resource contentions," in *RTAS, WIP Session*, 2016.

[14] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *CIT*, 2010.

[15] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson, "RTOS support for multicore mixed-criticality systems," in *18th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2012.

[16] J. Erickson, N. Kim, and J. Anderson, "Recovering from overload in multicore mixed-criticality systems," in *IPDPS*, 2015.

[17] RTCA SC-167, EUROCAE WG-12, Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[18] RTCA SC-205, EUROCAE WG-12, Software Considerations in Airborne Systems and Equipment Certification (DO-178C), 2012.

[19] J. E. Kim, T. Abdelzaher, and L. Sha, "Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model," in *RTAS*, 2015.

[20] ——, "Schedulability bound for integrated modular avionics partitions," in *DATE*, 2015.

[21] J. E. Kim, M. K. Yoon, R. Bradford, and L. Sha, "Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems," in *COMPSAC*, 2014.

[22] J. E. Kim, M. K. Yoon, S. Im, R. Bradford, and L. Sha, "Optimized scheduling of multi-IMA partitions with exclusive region for synchronized real-time multi-core system," in *DATE*, 2013.

[23] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele, "An isolation scheduling model for multicores," in *RTSS*, 2015.

[24] A. Burns and R. I. Davis, "Mixed criticality systems - a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

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

[26] J. Liu, K. Lin, and S. Natarajan, "Scheduling real-time, periodic jobs using imprecise results," in *RTSS*, 1987.

[27] W. Shih, W. Liu, J. Chung, and D. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *SIAM Journal of Computing*, vol. 20, no. 3, pp. 537–552, July 1991.

[28] J. Liu, W. Shih, K. Lin, R. Bettati, and J. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 84, no. 1, pp. 83–94, 1994.

[29] G. Koren and D. Shasha, "Skip-over: algorithms and complexity for overloaded systems that allow skips," in *RTSS*, 1992.

[30] M. Caccamo and G. Buttazzo, "Exploiting skips in periodic tasks for enhancing aperiodic responsiveness," in *RTSS*, 1997.

[31] J. P. Lehoczky, L. Sha, and J. Strosnider, "Enhancing aperiodic responsiveness in a hard real-time environment," in *RTSS*, 1987.

[32] L. Sha, J. Lehoczky, and R. Rajkumar, "Solutions for some practical problems in prioritised preemptive scheduling," in *RTSS*, 1986.

[33] R. Bril, J. Lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, no. 1, pp. 63–119, 2009.

[34] G. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. A survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[35] A. Melani, R. Mancuso, M. Caccamo, G. Buttazzo, G. Freitag, and S. Uhrig, "Extending Integrated Modular Avionic Systems on Multicore Platforms with Flexible Scheduling," University of Illinois at Urbana-Champaign, Tech. Rep., 2017. [Online]. Available: http://rtsl-edge.cs.illinois.edu/rmancuso/mc_ima_sched.pdf

[36] M. Chéramy, P. Hladik, and A. Déplanche, "SimSo: A simulation tool to evaluate real-time multiprocessor scheduling algorithms," in *WATERS*, 2014.

[37] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154.

[38] D. Locke, L. Lucas, and J. Goodenough, "Generic avionics software specification," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-008, 1990.