An Isolation Scheduling Model for Multicores

Pengcheng Huang, Georgia Giannopoulou, Rehan Ahmed, Davide B. Bartolini and Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland

***Abstract*—Efficiently exploiting multicore processors for real-time applications is challenging because jobs that run concurrently on different cores can interfere on shared resources, severely complicating precise timing analysis. We propose a new scheduling model called Isolation Scheduling (IS); IS provides a framework to exploiting multicores for real-time applications where tasks are grouped in classes. IS enforces mutually exclusive execution among different task classes, thus avoiding inter-class interference by construction. We show that IS encompasses several recent advances in real-time scheduling as special cases and we propose global and partitioned scheduling algorithms based on this model. Specific results are provided if the task classes correspond to different safety criticality levels.**

I. Introduction

There is a large gap between the requirements of real-time applications and what architectures of embedded processors offer today. On the one hand, real-time applications need predictability in order to enable safe operation based on worst-case execution time analysis. On the other hand, following the end of Dennard scaling [13], embedded processors increasingly feature a multicore architecture with shared resources (e.g., last-level cache, memory controller) in order to keep improving performance and efficiency. Figure 1 depicts an example of such an architecture.

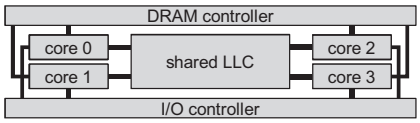


Figure 1. Sample typical multicore architecture with $m = 4$ cores that share last-level cache, DRAM controller and I/O controller.

To take advantage of multicore architectures, applications need to run jobs concurrently on different cores. Unfortunately, however, shared resources undermine predictability, since jobs that run concurrently pay unpredictable performance penalties due to contention in accessing shared resources.

Filling this gap is a challenging problem. Coarse-grain static partitioning in time and space, e.g., based on the DO-178B standard [1] for avionics or the ISO 26262 standard [2] for automotive systems, is an established technique for single-core safety-critical systems, but this approach can not be simply applied to multicore architectures; it would only allow one job to execute at any point in time if strictly applied. More fine-grained partitioning requires to individually control the access to each shared resource [32]. Finally, the approach of finding a global schedule and bounding the contention on shared resources at any time is only feasible with the knowledge of the detailed resource sharing behavior of all tasks, and it quickly becomes computationally intractable with an increasing number of tasks [15].

In this paper, we propose a new scheduling model that we call Isolation Scheduling (IS). IS is a practical model for efficiently scheduling real-time tasks on multicore processors, i.e., exploiting hardware parallelism and shared resources. To make the problem more tractable, IS makes an assumption about the tasks, i.e., we assume that tasks are partitioned into *task classes* that have exclusive access to the processor and the platform resources. This way, interference on shared resources is greatly reduced; inter-class interference is eliminated by construction, and only intra-class interference needs to be considered. Well-established methods [15, 18, 39] can be applied to bound / control this remaining interference.

Indeed, subdividing real-time tasks into classes provides key benefits in several contexts. For instance, *gang scheduling* [19, 24, 42] groups jobs (threads) that share information through fine-grained synchronization in the same class in order to reduce blocking times. Conversely, when jobs do not share information and there are well-defined task dependencies, communication takes place between classes in order to respect task dependencies, safely bound blocking times, and avoid concurrent access to shared memory. Furthermore, *server*-based scheduling [3] is an established approach for performance isolation among task classes with different timing requirements, e.g., for co-scheduling hard and soft real-time tasks or periodic and aperiodic tasks. Finally, in the context of *mixed-criticality systems* [37], tasks are grouped in classes of different safety criticality, and industrial standards [1, 2] pose strict requirements for isolating these classes in order to allow *independent certification* of criticality levels [11, 16, 36]. Isolation Scheduling guarantees that tasks of different criticality levels do not interfere on shared platform resources, and therefore, it allows for independent certification as well as a much simplified intra-class interference analysis.

Contribution and Outline. While recent work approached the idea of Isolation Scheduling from different angles (see Sec. III), to date this paper is the first to systematically formalize the IS model (Sec. II). We specifically analyze the IS model in the context of mixed-criticality scheduling and we propose two novel scheduling approaches, respectively based on fluid scheduling (Sec. IV and V) and on server-

1052-8725/15 $31.00 © 2015 IEEE
DOI 10.1109/RTSS.2015.21

based scheduling (Sec. VI). For each approach we provide scheduling algorithms and prove schedulability bounds, and we compare the approaches on scheduling large randomly-generated task sets (Sec. VII). Our results deliver a deeper understanding of the IS model and corresponding scheduling techniques, and suggest that the IS model is a useful and flexible abstraction for designing scheduling policies for systems that require strong isolation among task classes, such as mixed-criticality systems.

II. The Isolation Scheduling (IS) Model

The Isolation Scheduling (IS) model dynamically partitions a multicore processor in time between different task classes so that, at any time, only jobs of the same task class are allowed to execute on the platform. This strategy allows to partition the problem of bounding interference on shared resources to the single task classes; inter-class interference is completely disallowed. We target homogeneous multicore processors with m identical cores that share on-chip resources. Figure 1 shows an example of such an architecture.

The IS model supports real-time tasks τ, where each task periodically or sporadically instantiates single jobs. In addition, we assume that tasks are partitioned into K task classes $S = \{S_k \mid 1 \leq k \leq K\}$, where each task class S_k contains n_k tasks, i.e., $S_k = \{\tau_{i,k} \mid 1 \leq i \leq n_k\}$. For each task $\tau_{i,k}$ in task class S_k, the tuple $(T_{i,k}, D_{i,k}, C_{i,k})$ defines the period of the jobs (or their minimal inter-arrival time), their relative deadline, and their worst-case execution time (WCET). Additionally, we define the density $\delta_{i,k}$ and utilization $u_{i,k}$ of a task $\tau_{i,k}$ as

$$\delta_{i,k} = C_{i,k}/D_{i,k}, \quad u_{i,k} = C_{i,k}/T_{i,k} \tag{1}$$

Throughout the paper, we use k to index task classes, and i to index tasks within each task class. The set of task classes S is IS-schedulable if all tasks can meet their deadlines while respecting the IS-constraint of mutual exclusion between task classes. We will refine some of these notations when applying the IS model to mixed-criticality settings.

The IS model treats the multicore processor as a single resource that needs to be time-partitioned between the different scheduling classes. To give an intuition of how the model works, Figure 2 shows an example of an IS schedule.

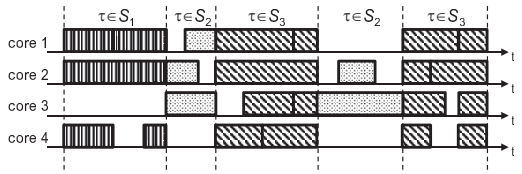


Figure 2. Example IS schedule with three task classes S_1, S_2 and S_3. Vertical lines mark the synchronous switching between task classes on all cores.

One way to relate the IS model to single-processor real-time scheduling is by comparing the synchronous task class switches (the vertical lines in Figure 2) with classic job preemption. When a single-core processor would switch between jobs, under the Isolation Scheduling model, all cores of the multicore processor synchronously switch between jobs of two task classes. In other words, classical job preemption is now lifted to synchronous switching between classes. This approach is different from single-core equivalence [32], which achieves task isolation by individually protecting each shared resource. An alternative way to look at the IS model is from a server perspective. Classical real-time servers [3] provide bounded capacities to task sets in a single processor setting. Recent proposals for multicore servers [10, 34] divide a multicore platform into multiple virtual platforms (servers) that could run in parallel. In Isolation Scheduling, the server acts globally on all cores, i.e., switching the resource allocation happens synchronously on all cores.

The IS model imposes synchronous class switching to bound resource interference, but remains general enough to support different scenarios. For instance, the model supports sporadic and periodic tasks; implicit, constrained, and arbitrary deadlines; preemptive and non-preemptive scheduling; static and dynamic time partitioning; and global and partitioned mapping of tasks to cores. Moreover, the implementation of an IS scheme depends on the specific application. For instance, if the switching between task classes can follow a static schedule, then a time-driven synchronization is appropriate; instead, if switching decisions are determined dynamically at run-time, then global synchronization mechanisms will be helpful. In Sec. IV to VI we analyze the schedulability and propose scheduling algorithms for some of these combinations.

Isolation Scheduling for Mixed-Criticality Systems. Mixed-criticality (MC) systems are a notable example of systems that require strong isolation between task classes. MC systems are real-time systems where tasks are partitioned into classes, commonly called *criticality levels*; different criticality levels have varying requirements in terms of correctness, assurance, and safety. To allow independent safety certification of criticality levels and avoid costly re-certification, task classes need to be strongly isolated [11, 16, 36]. While coarse-grained static partitioning grants such isolation for single-core systems, achieving a similar goal on multicore systems requires hardware and/or software mechanisms for guaranteeing interference-free or interference-bounded access of tasks to any shared resource, see for example [32]. Instead, the IS model takes advantage of the existence of well-defined task classes and avoids concurrent execution of jobs with different criticality by construction. In this way, different classes cannot contend on shared resources at all, criticality levels are completely isolated and they can be separately analyzed and certified. We use (particularly, in Sec. V and VI) mixed-criticality systems [37] as a case study to illustrate the usefulness of the IS model.

III. Background and Related Work

A. Mixed-criticality Systems — Formalization

When we discuss the IS model in the context of mixed-criticality systems, we focus on systems with two task classes (i.e., two criticality levels), as commonly assumed for simplicity [12]. For convenience, we summarize here the corresponding well-known model.

Task class S_{HI} only consists of tasks of high (HI) criticality; task class S_{LO} only includes LO criticality tasks. The execution time of HI criticality tasks is bounded on both criticality levels: $\forall \tau_i \in S_{\text{HI}}$, the term $C_i(\text{LO})$ denotes the *low execution time bound*, and $C_i(\text{HI})$ denotes the *high execution time bound*. The high execution time bound must be always guaranteed and is assumed to be more pessimistic than the low: $\forall \tau_i \in S_{\text{HI}}$ we require $C_i(\text{HI}) \geq C_i(\text{LO})$. LO criticality tasks only have one execution time bound, i.e., $C_i(\text{HI}) = C_i(\text{LO})$.

(2) defines the density $\delta_i(\chi)$ and utilization $u_i(\chi)$ of task τ_i as a function of its execution time bound $\chi \in \{\text{HI}, \text{LO}\}$.

$$\delta_i(\chi) := C_i(\chi)/D_i, \quad u_i(\chi) := C_i(\chi)/T_i \tag{2}$$

Similarly, (3) defines the density $\Delta_{\chi_1}^{\chi_2}$ and utilization $U_{\chi_1}^{\chi_2}$ of the task class S_{χ_1} with low ($\chi_2 = \text{LO}$) and high ($\chi_2 = \text{HI}$) execution time bounds.

$$\Delta_{\chi_1}^{\chi_2} := \sum_{\tau_i \in S_{\chi_1}} \delta_i(\chi_2), \quad U_{\chi_1}^{\chi_2} := \sum_{\tau_i \in S_{\chi_1}} u_i(\chi_2) \tag{3}$$

A dual criticality system is in LO *mode* as long as no HI criticality job overruns its low execution time bound $C_i(\text{LO})$; when at least one HI criticality job overruns its low execution time bound, the system switches to HI *mode*. The system must satisfy two schedulability requirements: 1) In LO mode, all jobs of LO and HI criticality are schedulable; and 2) in HI mode, all HI criticality jobs are schedulable. After a switch to HI mode, the system can switch back to LO mode under certain circumstances, e.g., when there are no more HI criticality jobs to be scheduled [8, 31].

B. Baseline Mixed-Criticality Scheduling (MC) Policies

We provide a short overview of two mixed-criticality (MC) scheduling policies, namely *partitioned EDF-VD* and *MC-Fluid*, as we will build upon them in Sec. V and VI. For a more extensive overview, we refer to the survey of Burns and Davis [12].

Partitioned EDF-VD. Baruah et al. [6] introduced EDF with virtual deadlines (EDF-VD) for uniprocessor implicit-deadline task sets, which was later extended to multicores under static task partitioning [8]. EDF-VD adapts classic preemptive EDF scheduling to ensure the schedulability of HI criticality tasks when the system switches to HI mode. To achieve this goal, EDF-VD assigns *virtual deadlines* to HI criticality jobs, i.e., to jobs of tasks in the S_{HI} class. The virtual deadline is computed by multiplying the original deadline by a fixed factor $x \in (0, 1]$. In LO mode, jobs are scheduled according to EDF, using the original deadlines for LO criticality jobs and the virtual deadlines of HI criticality jobs. Since the virtual deadlines are down-scaled, HI criticality jobs will have some slack to "catch up" upon switching to HI mode. In case the system switches to HI mode, all LO criticality jobs are dropped and HI criticality jobs are scheduled with EDF using their original deadlines.

Baruah et al. [6] proved that, in LO mode, all deadlines (for HI criticality tasks, virtual deadlines) will be met if:

$$x \geq (U_{\text{HI}}^{\text{LO}})/(1 - U_{\text{LO}}^{\text{LO}}) \tag{4}$$

Moreover, Huang et al. [22] proved that there will be no deadline violations for HI criticality tasks during switch to HI mode and during HI mode operation if all HI criticality tasks finish by their virtual deadlines in LO mode and

$$\sum_{\tau_i \in S_{\text{HI}}} \frac{u_i(\text{HI})}{u_i(\text{LO}) + (1 - x)} \leq 1 \tag{5}$$

We will use the results from (4) and (5) in our IS-based scheduling policies, see Sec. V and VI.

MC-Fluid & DP-Fair. Lee et al. [26] extended a well known optimal multicore scheduling technique, DP-Fair [27], to dual-criticality task systems by proposing MC-Fluid. DP-Fair is a fluid based scheduling technique, which enforces proportional progress of all tasks within dedicated system slices. DP-Fair only requires a minimal set of rules for achieving proportional progress, covering many other scheduling techniques as special cases, e.g., the original P-Fair algorithm [7]. Conceptually similar to EDF-VD, MC-Fluid runs the system in two modes, with the deadlines of HI criticality tasks shortened in LO mode. In both LO and HI system modes, tasks are scheduled by DP-Fair. Lee et al. [26] provided a corresponding schedulability test and theoretically investigated the performance of MC-Fluid.

C. Task Class Isolation in MC Scheduling

Initial research on isolating task classes on multicores did not explicitly address interference between task classes when jobs concurrently access shared resources, i.e., they imply that it can be bounded. For instance, Anderson et al. [4] and Mollison et al. [28] adopt different strategies (partitioned EDF, global EDF, cyclic executive) for different task classes and use a bandwidth reservation server for timing isolation between classes. However, interference analysis for multiple shared resources is a very challenging task by itself. In fact, estimating response time bounds under contention may be even impossible for MC systems, because a certification authority for higher criticality tasks does not necessarily possess information on the behavior of lower criticality tasks that are co-hosted on the same platform.

Subsequently, researchers acknowledged the problem of inter-class interference and proposed mechanisms for criticality-aware arbitration of shared resources, with the

objective of statically bounding interference from lower to higher criticality tasks. Yun et al. [40] and Flodin et al. [14] proposed a software-based memory throttling mechanism (with predefined [40] or dynamically allocated [14] per-core budgets) to explicitly control interference on a shared memory controller. Goossens et al. [20], Paolieri et al. [29] proposed hardware modifications to a shared memory controller for mixed hard and soft real-time systems. Recent works [17, 30, 39, 41] proposed partitioning data to disjoint DRAM banks in order to minimize inter-core interference. Tămaş-Selicean and Pop [36] present optimization methods for time-triggered partition scheduling on heterogeneous multicores that comply with the ARINC-653 standard [5]; they assume that the platform provides both spatial and temporal partitioning that enforce enough isolation between task classes. Similarly, Kim et al. [25] rely on ARINC-653 compliance to devise a method for conflict-free I/O transactions. Finally, Tobuschat et al. [35] implemented virtualization and monitoring mechanisms to provide independence among flows of different criticality in networks-on-chips. These mechanisms and policies ensure sufficient isolation among criticality levels, but they suffer from poor flexibility, e.g., memory budgets [40] cannot change dynamically if the resource demand changes, and they may require special hardware support, which is not widely available.

Finally, Giannopoulou et al. [16] and Burns et al. [11] recently proposed scheduling strategies that sidestep the need for fine-grained shared resource arbitration. The key idea is to only permit tasks of the same criticality (i.e., from the same class) to execute concurrently. Based on this insight, the scheduling policies they propose avoid resource interference among task classes, exploit multiple cores, and only suffer a limited schedulability loss to enforce time-partitioning among task classes. The Isolation Scheduling model we propose includes both these policies as special cases; additionally, we propose novel policies built from scratch upon the IS model.

IV. The IS-DP-Fair Scheduling Policy

Optimal scheduling for multicores is provided, for both periodic and sporadic task sets, by the DP-Fair family of algorithms [27], which originated from P-Fair [7]. These algorithms allow inter-class contention and inter-class use of shared resources, as discussed in Sec. I and III. We tackle this shortcoming by "porting" DP-Fair into the IS model, i.e., we extend DP-Fair with the *IS constraint* of strong isolation between task classes. We call this new algorithm IS-DP-Fair and we present its schedulability analysis, focusing on periodic tasks. We show that, for tasks with implicit deadlines, IS-DP-Fair is optimal in terms of schedulability among all possible schedulers based on the IS model; we quantify the schedulability loss of IS-DP-Fair compared to non-IS schedulers.

Algorithm 1: IS-DP-Fair

1 Let t_1, t_2, $\cdots$ be the list (assuming no duplicates) of all arrival times and deadlines of the jobs to be scheduled; subdivide time into basic slices σ_j, separated by the t_j time points:

$$\sigma_j = [t_j, t_{j+1}), \text{ with length } L_j$$

2 Subdivide each σ_j into K subslices: $\sigma_j = \{\sigma_{j,k} | 1 \leq k \leq K\}$. The length of $\sigma_{j,k}$ is

$$L_{j,k} = \max \left\{ L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{L_j}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\}$$

3 $\forall i, j, k$ allocate an execution budget $\delta_{i,k} L_j$ for task $\tau_{i,k}$ in subslice $\sigma_{j,k}$.

4 Schedule tasks within each subslice with DP-Fair.

A. The IS-DP-Fair Algorithm

Algorithm 1 outlines the IS-DP-Fair algorithm. Similarly to DP-Fair, we first partition time into slices: given the list of all arrival times and deadlines without duplicates, a slice σ_j is the time span between two consecutive such instants.

As a second step, we subdivide each such slice σ_j into K consecutive subslices; each subslice $\sigma_{j,k}$ of length $L_{j,k}$ (see Algorithm 1, step 2) is used to exclusively host task class S_k.

Third, we allocate the execution budget to all tasks. The idea is to enforce proportional progress of all tasks (from different task classes) within each slice: each task $\tau_{i,k}$ is exclusively assigned an execution budget $\delta_{i,k} L_j$ in subslice $\sigma_{j,k}$ within σ_j. Within each subslice $\sigma_{j,k}$, tasks are assigned iteratively in a greedy way: each task is assigned to a processor that has a non-empty (i.e., at least one task was already assigned to it) and non-full (i.e., there is free capacity) subslice $\sigma_{j,k}$. If no such processor exists, the task is assigned to a processor that has empty $\sigma_{j,k}$. If the available capacity in the subslice $\sigma_{j,k}$ for the chosen processor is not enough to serve all the execution budget of task $\tau_{j,k}$, then the budget is split and the excess is allocated to the next available processor. This process continues until all tasks are allocated.

After allocation, we know exactly when and on which processor the budget for each task will be available. At runtime, any task executes whenever its budget becomes available; as long as the task does not arrive, the corresponding budget is idled.

B. Schedulability Analysis of IS-DP-Fair

Theorem 1 gives an exact schedulability test for using IS-DP-Fair to schedule a task set S with K task classes S_k, with $1 \leq k \leq K$, on a multicore architecture with m identical cores.

Theorem 1. Task set S is schedulable with IS-DP-Fair *iff*

$$\sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq 1 \quad (6)$$

Proof: According to the IS-DP-Fair algorithm, the arrival time t_a and the absolute deadline t_d ($d > a$) of any job of any task $\tau_{i,k}$ from task class S_k coincide with the start or the end of a (possibly different) slice. There might be multiple slices σ_j between t_a and t_d, where $a \leq j < d$. Task $\tau_{i,k}$ is guaranteed to meet its deadline if its execution budget is satisfied across these slices.

Within each slice σ_j, any task $\tau_{i,k}$ receives, by construction, an execution budget of $\delta_{i,k} L_j$ for each subslice $\sigma_{j,k}$ and, since a task cannot run in parallel with itself, we find

$$L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\} \leq L_{j,k} \quad (7)$$

In addition, since all tasks from S_k must be schedulable by DP-Fair within $\sigma_{j,k}$, i.e., the cumulative allocated budget must be less than or equal to the available processor time from all cores, we conclude

$$L_j \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \leq m L_{j,k} \quad (8)$$

Combining the two bounds from (7) and (8), we get

$$L_{j,k} \geq \max \left\{ L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{L_j}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \quad (9)$$

Setting $L_{j,k}$ to the minimum value that satisfies (9) guarantees schedulability within each subslice. Finally we consider the additional constraint

$$\begin{aligned} & \sum_{S_k \in S} L_{j,k} \leq L_j \\ \iff & \sum_{S_k \in S} \max \left\{ L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{L_j}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq L_j \\ \iff & \sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq 1 \quad (10) \end{aligned}$$

Therefore, the test of (6) is sufficient for schedulability with IS-DP-Fair and, if the test of (6) fails, then the task set S cannot be scheduled with IS-DP-Fair. Thus, the test as specified in (6) is exact, i.e., both sufficient and necessary. ■

After providing the schedulability test of Theorem 1, we show in Theorem 2 that the IS-DP-Fair algorithm is optimal for any task set $S = \{S_k\}$ with implicit deadlines under the IS constraint, i.e., under isolation of task classes.

Theorem 2. IS-DP-Fair is optimal in terms of schedulability for task sets with *implicit deadlines* under the IS constraint.

Proof: We prove Theorem 2 by showing that, whenever IS-DP-Fair fails, no other scheduling solution exists. Since the schedulability test of Theorem 1 is exact, assuming that IS-DP-Fair fails for task set $S = \{S_k\}$ implies

$$\sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} > 1 \quad (11)$$

For the purpose of contradiction, assume that S is still schedulable by some other scheduling algorithm Λ.

Consider the hyperperiod of the tasks of S in the case when all tasks initially arrive at time zero. Let T_{hyper} denote the duration of the hyperperiod and $T_{hyper}.k$ denote the total duration of the subslices allocated to task class S_k within the hyperperiod. A task $\tau_{i,k}$ in S_k cannot run in parallel with itself and it must execute for $\delta_{i,k} T_{hyper}$ within T_{hyper}. Therefore,

$$\begin{aligned} & \forall \tau_{i,k} \in S_k : \ T_{hyper}.k \geq \delta_{i,k} T_{hyper} \\ \iff & T_{hyper}.k \geq T_{hyper} \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\} \end{aligned}$$

Since we assumed that S is schedulable by Λ, then all tasks from S_k meet their deadlines within T_{hyper}, implying

$$\begin{aligned} & m T_{hyper}.k \geq \sum_{\tau_{i,k} \in S_k} \delta_{i,k} T_{hyper} \\ \iff & T_{hyper}.k \geq \frac{T_{hyper}}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \end{aligned}$$

Within the hyperperiod, the total fraction of all slices allocated to any task class cannot be more than T_{hyper}. Therefore:

$$\begin{aligned} T_{hyper} \geq & \sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k} T_{hyper}\}, \frac{T_{hyper}}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \\ \iff & \sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq 1 \end{aligned} \quad (12)$$

(11) contradicts (12), so no algorithm Λ can schedule S. ■

While IS-DP-Fair is optimal for implicit deadlines, the same property does not hold for the case of a task set S with constrained deadlines, as Theorem 3 states.

Theorem 3. IS-DP-Fair is not optimal for task sets with constrained deadlines under the IS constraint.

Proof: We prove Theorem 3 by showing a counterexample. Consider a task set S with two task classes S_1 and S_2, each containing a single task, respectively $\tau_1 = (2, 1, 1)$[1] and $\tau_2 = (2, 2, 1)$. Trying to schedule S on a dual-core processor fails the schedulability test of Theorem 1. However, S is in

[1]The tuple defines the period, relative deadline, and worst-case execution time of the task, see Sec. II.

fact schedulable. Since each class only contains one task, we can reduce the problem to a single core scheduling problem and apply fixed-priority scheduling (τ_1 with higher priority). Then it is straightforward to see, through conventional response time analysis, that S is schedulable. ■

C. Schedulability Loss of IS-DP-Fair

Finally, it is important to quantify the loss of schedulability due to enforcing the IS constraint, compared to allowing inter-class interference. Theorem 4 provides a tight bound on the speedup required to enforce isolation.

Theorem 4. Any task set S schedulable with DP-Fair (by removing the IS constraint) is schedulable by IS-DP-Fair under the IS constraint on a platform that is $\min\{K, m\}$ times faster. This speedup bound is tight.

Proof: Due to space limitations, please refer to [23] for detailed proof. ■

Though IS-DP-Fair is not optimal in terms of schedulability for task sets with constrained deadlines, we find that it is optimal with respect to the speedup bound of Theorem 4.

Theorem 5. Under the IS constraint, no scheduler can achieve a speedup bound better than $\min\{K, m\}$ compared to an optimal scheduling algorithm that ignores the IS constraint.

Proof: Proof is provided in [23]. ■

While, in general, there is a cost for enforcing the IS constraint (as Theorem 4 and 5 show), there is no such cost to pay under suitable assumptions, as Corollary 1 shows.

Corollary 1. If a task set S is schedulable by ignoring the IS constraint and if it satisfies the condition

$$\forall S_k \in S: \quad \frac{\max\limits_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}}{\operatorname{avg}\limits_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}} \leq \frac{|S_k|}{m} \tag{13}$$

where the operator avg computes the average of its arguments, then S is also schedulable with IS-DP-Fair.

Proof: Proof is provided in [23]. ■

Essentially Corollary 1 states that the schedulability loss due to isolation decreases, as the variation in density across tasks within task classes decreases and, as the number of tasks within task classes increases.

V. IS-DP-Fair for Mixed-Criticality Systems

An immediate application of IS-DP-Fair is scheduling mixed-criticality (MC) systems on multicores. Thanks to the IS model, IS-DP-Fair ensures isolation between criticality levels. While all the results of Sec. IV apply, MC systems have some additional peculiarities (see Sec. III), e.g., they can switch between criticality modes. For this reason, we extend IS-DP-Fair for MC systems; we call this new algorithm MC-IS-Fluid. An extension of the DP-Fair algorithm for MC systems exists for multiprocessors without considering isolation. We outlined this algorithm, known as MC-Fluid [26], in Sec. III-B. The key idea, borrowed from Baruah et al. [6], is to shorten the deadlines of HI criticality tasks in LO mode, in order to shift demand from HI to LO mode and improve schedulability. We adopt the same technique in MC-IS-Fluid.

Algorithm 2: MC-IS-Fluid

1 For all HI criticality tasks $\tau_i \in S_{\text{HI}}$, compute the shortened deadline $D'_i = xD_i$, to be used in LO mode; the shortening factor $x : 0 < x \leq 1$ is:

$$x = \frac{\max\left\{\max\limits_{\tau_i \in S_{\text{HI}}} \{\delta_i(\text{LO})\},\ \frac{1}{m}\Delta_{\text{HI}}^{\text{LO}}\right\}}{1 - \max\left\{\max\limits_{\tau_i \in S_{\text{LO}}} \{\delta_i(\text{LO})\},\ \frac{1}{m}\Delta_{\text{LO}}^{\text{LO}}\right\}} \tag{14}$$

2 In LO mode, set the density of all HI criticality tasks $\tau_i \in S_{\text{HI}}$ to $\delta_i(\text{LO})/x$; for LO criticality tasks $\tau_i \in S_{\text{LO}}$ use the density $\delta_i(\text{LO})$. In LO mode, schedule all tasks by IS-DP-Fair.

3 If any HI criticality task overruns its LO level WCET, first conclude the current slice; then, switch to HI mode by terminating all LO criticality tasks and restoring the original deadlines of HI criticality tasks. Schedule the remaining HI criticality tasks with DP-Fair (densities for HI criticality tasks set by Lemma 1).

A. The MC-IS-Fluid Algorithm

Algorithm 2 outlines the MC-IS-Fluid algorithm for a dual-criticality task set $S = \{S_{\text{HI}}, S_{\text{LO}}\}$. We compute shortened deadlines for the HI criticality tasks, similarly to MC-Fluid. In LO mode, we schedule the system with IS-DP-Fair, using the original deadlines for LO criticality tasks and the shortened deadlines for HI criticality tasks. After a switch to HI mode, all LO criticality tasks are dropped and we schedule the remaining HI criticality tasks with a DP-Fair compatible scheduling technique, using the original deadlines. While using a similar approach to MC-Fluid, MC-IS-Fluid has two main differences: 1) we shorten the deadlines of HI criticality tasks uniformly, so we are able to provide a closed-form schedulability test; and 2) we show (see Lemma 2) that greedily shortening the deadlines of HI criticality tasks is optimal for schedulability in HI mode.

B. Schedulability Analysis for MC-IS-Fluid

In LO mode, MC-IS-Fluid uses shortened deadlines $D'_i = xD_i$, with $0 < x \leq 1$ (see Algorithm 2), for all HI criticality tasks $\tau_i \in S_{\text{HI}}$. Therefore, the density of each task $\tau_i \in S_{\text{HI}}$ in LO mode increases to $\delta_i(\text{LO})/x$. MC-IS-Fluid does not shorten the deadlines of LO criticality tasks; so, $\forall \tau_i \in S_{\text{LO}}$, the density is $\delta_i(\text{LO})$. We can use the schedulability test of Theorem 1, by simply using the shortened deadlines for HI criticality tasks, to test schedulability in LO mode.

Finding a closed-form schedulability test for MC-IS-Fluid in HI mode is less trivial because, in general, we do not know when the system will switch to HI mode. If, at mode switch, a partially executed HI criticality job is *carried over* to HI mode, we need to know the remaining execution requirement of this job and the time until its actual deadline in order to bound the maximum task density in HI mode. Let us denote such maximum density of a HI criticality task τ_i in HI mode as $\delta_i^{\max}(\text{HI})$. Formally, we establish the following result.

Lemma 1. For any HI criticality task τ_i, $\delta_i^{\max}(\text{HI}) = \max\left\{\dfrac{\delta_i(\text{HI}) - \delta_i(\text{LO})}{1-x}, \delta_i(\text{HI})\right\}$.

Proof: Consider $\tau_i \in S_{\text{HI}}$. If there is no carry-over job from this task in HI mode, then we only need to consider jobs of τ_i that arrive after the mode switch. In this case the maximum density of τ_i in HI mode equals to $\delta_i(\text{HI})$.

Otherwise, we need to consider the carry-over job. Assume that the mode switch happens $t^\star$ after the arrival of a job of τ_i, where $0 \leq t^\star \leq xD_i$. Since, according to Algorithm 2, the mode switch coincides with the end of one slice in LO mode, then the density of this carry-over job in HI mode is:

$$\delta_i^*(\text{HI}) = \frac{C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x} t^\star}{D_i - t^\star} \tag{15}$$

Since t^* can only vary in the interval $[0, xD_i]$, we need to find the maximum of $\delta_i^*(\text{HI})$ within this interval. To do so, in (16) we compute $d\,\delta_i^*(\text{HI})/dt^\star$, i.e., the first order derivative of $\delta_i^*(\text{HI})$ with respect to t^*:

$$\begin{aligned}
&\frac{d\,\delta_i^*(\text{HI})}{dt^\star} = \frac{d\left(\frac{C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x} t^\star}{D_i - t^\star}\right)}{dt^\star} \\
=&\frac{1}{(D_i - t^\star)^2}\left(-\frac{\delta_i(\text{LO})}{x}(D_i - t^\star) + C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x} t^\star\right) \\
=&\frac{1}{(D_i - t^\star)^2}\left(-\frac{\delta_i(\text{LO})}{x} D_i + C_i(\text{HI})\right)
\end{aligned} \tag{16}$$

Looking at (16), the sign of $d\,\delta_i^*(\text{HI})/dt^\star$ does not change within the interval $[0, xD_i]$. Therefore, the maximum of $\delta_i^*(\text{HI})$ will be at one of the extremes of the interval, according to whether $\delta_i^*(\text{HI})$ is increasing or decreasing (respectively, if $d\,\delta_i^*(\text{HI})/dt^\star$ is positive or negative). Since the sign of (16) is determined by the second factor, we can look at two cases:

- When $\delta_i(\text{LO})/x \leq \delta_i(\text{HI})$, (16) is non-negative and the maximum density of the carry-over is when $t^\star = xD_i$:

$$\begin{aligned}
&\max_{t^\star = xD_i}\{\delta_i^*(\text{HI})\} \\
=&\frac{C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x} xD_i}{D_i - xD_i} = \frac{\delta_i(\text{HI}) - \delta_i(\text{LO})}{1-x} \\
\geq& \frac{\delta_i(\text{HI}) - x\delta_i(\text{HI})}{1-x} = \delta_i(\text{HI})
\end{aligned} \tag{17}$$

- When $\delta_i(\text{LO})/x > \delta_i(\text{HI})$, (16) is negative and the maximum density of the carry-over is when $t^\star = 0$:

$$\begin{aligned}
&\max_{t^\star = 0}\{\delta_i^*(\text{HI})\} \\
=&\delta_i(\text{HI}) = \frac{\delta_i(\text{HI}) - x\delta_i(\text{HI})}{1-x} > \frac{\delta_i(\text{HI}) - \delta_i(\text{LO})}{1-x}
\end{aligned} \tag{18}$$

Putting the two cases together, we get Lemma 1. ∎

Using Lemma 1, we can formally determine a schedulability test for HI mode (Theorem 6).

Theorem 6. A dual-criticality task set $S = \{S_{\text{HI}}, S_{\text{LO}}\}$ is schedulable in HI mode under MC-IS-Fluid *iff*

$$\max\left\{\max_{\tau_i \in S_{\text{HI}}}\{\delta_i^{\max}(\text{HI})\},\ \frac{1}{m}\sum_{\tau_i \in S_{\text{HI}}} \delta_i^{\max}(\text{HI})\right\} \leq 1 \tag{19}$$

Proof: Theorem 6 directly follows from Lemma 1 and the DP-Fair schedulability test [27]. ∎

C. Optimal Greedy Choice of x

So far, our analysis did not assume any particular choice of x. With Lemma 2, we show now that setting x according to (14) in MC-IS-Fluid is optimal in terms of schedulability.

Lemma 2. Whenever a dual-criticality task $S = \{S_{\text{HI}}, S_{\text{LO}}\}$ is schedulable with MC-IS-Fluid for some choice of x, it is also schedulable when x is set according to (14).

Proof: Based on Theorem 1, in order to guarantee schedulability under IS-DP-Fair in LO mode, we have:

$$\begin{aligned}
&\frac{1}{x}\max\left\{\max_{\tau_i \in S_{\text{HI}}}\{\delta_i(\text{LO})\}, \frac{\Delta_{\text{HI}}^{\text{LO}}}{m}\right\} \\
+&\max\left\{\max_{\tau_i \in S_{\text{LO}}}\{\delta_i(\text{LO})\}, \frac{\Delta_{\text{LO}}^{\text{LO}}}{m}\right\} \leq 1
\end{aligned} \tag{20}$$

Now, suppose that there exists some x' which leads to a schedulable system in both LO and HI modes. Then, in order to guarantee LO mode schedulability, we must have that $x' \geq x$. According to Lemma 1 and Theorem 6, if we choose x instead of x', then the maximum task density in HI mode will not increase and the system remains schedulable in HI mode. Therefore, setting x according to (14) is optimal. ∎

Finally, with Theorem 7, we summarize our analysis into a complete schedulability test.

Theorem 7. A dual-criticality task $S = \{S_{\text{HI}}, S_{\text{LO}}\}$ is schedulable with MC-IS-Fluid under the IS constraint *iff* $0 < x \leq 1$, with x set by (14), and (19) is satisfied.

Proof: Theorem 7 directly follows from Theorem 1, Theorem 6 and Lemma 2. ∎

VI. The IS-Server and MC-IS-Server Policies

A possible issue with all *fluid* scheduling algorithms, including IS-DP-Fair (Sec. IV) and MC-IS-Fluid (Sec. V), is the large number of preemptions, migrations and in our case, synchronous task class switches, which can be costly. To address this issue, we explore a different server-based scheduling strategy for the IS model; we refer to this new strategy as IS-Server. We first analyze MC-IS-Server, a variant of IS-Server for mixed-criticality (MC) systems and then we show that IS-Server is really a special case of MC-IS-Server, without the mixed-criticality requirements (Sec. VI-C). Notice that existing server methods for multi-core platforms [10, 34] assume that tasks are grouped into multiple virtual platforms (servers) that could run in parallel. Such methods violate the Isolation Scheduling constraint we assume in this paper. Instead, we propose global servers, one for each task class, which have exclusive access to the underlying platform. Compared to the time-triggered approach proposed in [18], we generalize the server method to incorporate EDF-based (rather than static) scheduling.

Similarly to partitioned EDF-VD [8], for MC-IS-Server we focus on dual-criticality implicit-deadline task sets, which need strong isolation. MC-IS-Server is a *partitioned* scheduling algorithm, i.e., tasks are not allowed to migrate across cores. The algorithm has two phases: 1) In the first phase, we partition tasks using a MIQCP (Mixed Integer Quadratically Constrained Programming) formulation; in this phase, we make sure to guarantee schedulability in LO and HI modes on all cores. 2) After partitioning, we employ a search strategy to find a periodic server schedule of maximal period such that all tasks meet their deadlines. All cores perform the same server schedule, thus guaranteeing mutual exclusion among criticality levels / task classes.

A. Task Partitioning

The partitioning phase of MC-IS-Server assigns tasks to cores in order to guarantee both LO and HI criticality behavior. To do so, we need to make sure that schedulability conditions specified in Sec. III, (4) and (5), hold for *each* core. Figure 3 formulates the partitioning phase as a MIQCP problem. Similarly to other MC schedulers (see Sec. III-B), in LO mode we scale the deadlines of HI criticality tasks assigned to core j by a factor x_j; note that this is in contrast to MC-IS-Fluid (Sec. V), which adopts a single scaling factor x for all cores. The objective function that we want to minimize is the maximum LO mode utilization across all cores. Constraint (21) guarantees that each task is assigned to exactly one core and Constraint (22) bounds the scaling factors $x_j, 1 \leq j \leq m$, so that HI mode schedulability of the system is guaranteed, as required by (5). While solving the MIQCP problem is computationally expensive, this phase only needs to run offline. In practice, the time the optimizer takes to run is acceptable on modern hardware (99% of the evaluated task sets in Sec. VII were optimally partitioned in less than 100 seconds on a quad-core Core-i7 Haswell platform). As alternative to the MIQCP formulation, the partitioning phase could employ a task partitioning algorithm, for instance MC-Partition [8].

Variables:

x_j is the deadline scaling factor for HI tasks on core j

$$\alpha_{i,j} = \begin{cases} 1, & \text{if task } \tau_i \text{ is assigned to core } j \\ 0, & \text{otherwise} \end{cases}$$

Objective:

$$\text{Minimize} \max_{1 \leq j \leq m} \left\{ \sum_{\substack{\tau_i \in S_{\text{HI}} \\ \alpha_{i,j}=1}} u_i(\text{LO})/x_j + \sum_{\substack{\tau_i \in S_{\text{LO}} \\ \alpha_{i,j}=1}} u_i(\text{LO}) \right\}$$

Constraints:

$$\sum_{1 \leq j \leq m} \alpha_{i,j} = 1 \qquad \forall \tau_i \in \{S_{\text{HI}}, S_{\text{LO}}\} \tag{21}$$

$$\sum_{\substack{\tau_i \in S_{\text{HI}} \\ \alpha_{i,j}=1}} \left(\frac{u_i(\text{HI})}{u_i(\text{LO}) + 1 - x_j} \right) \leq 1 \qquad \forall\, 1 \leq j \leq m \tag{22}$$

Figure 3. Mixed Integer Quadratically Constrained Programming (MIQCP) formulation for the partitioning phase.

B. Server Scheduling

Once the partitioning phase is done, we apply hierarchical scheduling using K global servers, one for each task class S_k ($1 \leq k \leq K$). Since we focus on dual-criticality systems, we consider two servers: one for HI criticality tasks (from class S_{HI}) and one for LO criticality tasks (from class S_{LO}). To enforce the IS constraint, only one global server can be active at any time.

When the system is in LO mode, MC-IS-Server schedules tasks within each global server according to the EDF policy, using the shortened deadlines (scaled by x_j on core j) for HI criticality tasks. To schedule the global servers, we consider the whole multicore as a single TDMA resource [21, 38]. MC-IS-Server periodically assigns the TDMA resource (i.e., the whole multicore) to the HI and the LO server according to a predefined timing pattern. This pattern recurs with a period P, which is common to both global servers. Each server is assigned the multicore for a fraction of the period P, proportional to its execution budget. The execution slots for HI and LO tasks are called ℓ_{HI} and ℓ_{LO}, respectively. We assume full system utilization, i.e., $P = \ell_{\text{HI}} + \ell_{\text{LO}}$. Note that, as alternative to the TDMA server schedule, other server strategies (e.g., periodic resource servers [33]) could be used.

Upon switch to HI mode, MC-IS-Server drops all LO criticality tasks (effectively disabling the associated server) and it schedules the remaining HI criticality tasks with partitioned EDF using their original (non-shortened) deadlines.

Schedulability in HI mode is ensured by the constraints of the partitioning phase (see Sec. VI-A).

Recall that the LO mode schedulability in our partitioning phase is only guaranteed with best-effort, i.e. through an optimization goal. To make a hard guarantee in MC-IS-Server, we now determine a feasible period P and the server budgets ℓ_{HI} and ℓ_{LO} through a search method. The goal here is to maximize the period P in order to minimize the synchronous task class switches, which can be costly. To illustrate how MC-IS-Server finds the period P and the budgets, ℓ_{HI} and ℓ_{LO}, we first define some useful terms.

*Task demand bound function (*tdbf). The function $\text{tdbf}(\tau_i, t)$ is the maximum execution demand of task τ_i over any time interval of size t [9]:

$$\text{tdbf}(\tau_i, t) = \max\left\{\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1, 0\right\} \cdot C_i$$

*Per-Core Class Demand Bound Function (*pc_cdbf). The function $\text{pc\_cdbf}(S_k, t, j)$ is the maximum aggregate execution demand of all tasks τ_i in task class S_k, which run on core j over any time interval of size t:

$$\text{pc\_cdbf}(S_k, t, j) = \sum_{\substack{\tau_i \in S_k \\ \tau_i \text{ on core } j}} \text{tdbf}(\tau_i, t)$$

*Class Demand Bound Function (*cdbf). The function $\text{cdbf}(S_k, t)$ is the maximum aggregate execution demand of task class S_k on all cores, over any time interval of size t:

$$\text{cdbf}(S_k, t) = \max_{0 \le j \le m} \{\text{pc\_cdbf}(S_k, t, j)\}$$

*Supply Bound Function (*sbf). The function $\text{sbf}(S_k, t)$ for the server associated with task class S_k is the minimum service (execution cycles) that the server provides over any time interval of size t:

$$\text{sbf}(S_k, t) = \left\lfloor \frac{t}{P} \right\rfloor \ell_{S_k} + \max\left\{t - (P - \ell_{S_k}) - \left\lfloor \frac{t}{P} \right\rfloor \cdot P,\ 0\right\}$$

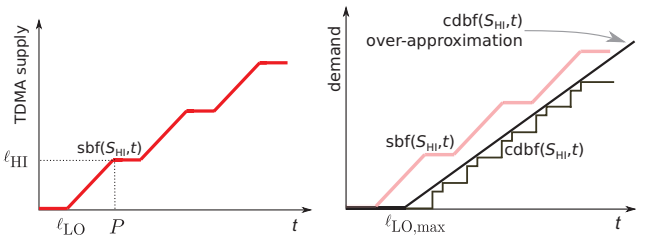
For the global servers to ensure schedulability of all tasks in LO mode, the following conditions must hold:

$$\text{sbf}(S_{\text{HI}}, t) \ge \text{cdbf}(S_{\text{HI}}, t), \qquad \forall t \ge 0, \tag{23}$$

$$\text{sbf}(S_{\text{LO}}, t) \ge \text{cdbf}(S_{\text{LO}}, t), \qquad \forall t \ge 0. \tag{24}$$

Note that, in LO mode, the demand bound function of HI criticality tasks is computed using the shortened deadlines.

We illustrate how MC-IS-Server finds suitable values for P, ℓ_{HI}, and ℓ_{LO} through an example. Figure 4 shows an example supply bound function of the HI server ($\text{sbf}(S_{\text{HI}}, t)$, Figure 4a) and the corresponding demand bound function of task class S_{HI} ($\text{cdbf}(S_{\text{HI}}, t)$, Figure 4b). For both task classes (Figure 4 illustrates only the case for S_{HI}), we determine linear over-approximations of the two class demand bound functions (in Figure 4b, the black solid line is the one for S_{HI}). When dealing with task class S_{HI}, the intersection of the over-approximation of $\text{cdbf}(S_{\text{HI}}, t)$ with the x-axis



(a) Supply bound function for the HI global server. (b) Demand bound functions for task class S_{HI}.

Figure 4. Supply and demand bound functions for the HI global server.

(i.e., for demand = 0) gives us the maximum budget we can allocate to the LO server $\ell_{\text{LO,max}}$; respectively, the approximation of $\text{cdbf}(S_{\text{LO}}, t)$ determines $\ell_{\text{HI,max}}$. We perform binary search on $\ell_{\text{LO,max}}$ to jointly adjust the intersection points and slopes of the two approximation curves, such that $P = \ell_{\text{HI}} + \ell_{\text{LO}}$ is maximized and schedulability is guaranteed. For a more detailed explanation of the search method, please refer to [23].

C. The IS-Server Policy for non Mixed-Critical Systems

IS-Server is a variant of MC-IS-Server for non mixed-critical systems. The algorithm is mostly the same, it becomes a little simpler because we do not need to consider switching to HI mode. In particular, the task partitioning phase becomes simpler because of the absence of deadline scaling factors. In the partitioning phase, IS-Server looks for task to core assignments ($\alpha_{i,j}$) that minimize the objective function (25):

$$\max_{1 \le j \le m} \max\left\{\sum_{\substack{\tau_{i,k} \in S_1 \\ \alpha_{i,j}=1}} u_{i,k}, \sum_{\substack{\tau_{i,k} \in S_2 \\ \alpha_{i,j}=1}} u_{i,k}\right\} \tag{25}$$

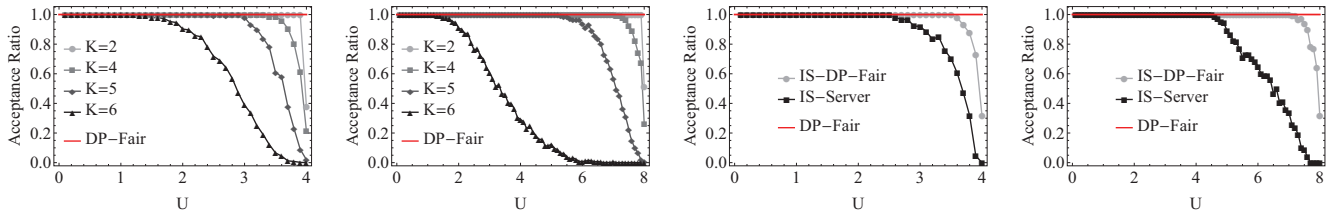
subject to the partitioning constraint (21). After partitioning, the search strategy for server parameters is the same as for MC-IS-Server. Note that IS-Server minimizes the *maximum* of the utilizations of the two task classes across cores. Instead, MC-IS-Server minimizes the *sum* of the utilizations of S_{LO} and S_{HI}. We choose a different objective function for IS-Server because we found that it gives better results.

VII. Experimental Evaluation

In the following, we evaluate and compare the performance of IS-DP-Fair, MC-IS-Fluid (Sec. IV and V), IS-Server and MC-IS-Server (Sec. VI) in terms of schedulability, synchronous switches between task classes, and task migrations. Additionally, we provide comparisons with state-of-the-art scheduling techniques.

A. Random Task Set Generation

We synthetically generate implicit-deadline periodic task sets at different system utilization points. For creating basic (non mixed-criticality) IS task classes, we generate tasks in the following manner:



(a) IS-DP-Fair: Impact of number of classes on schedulability, 4 cores

(b) IS-DP-Fair: Impact of number of classes on schedulability, 8 cores

(c) Schedulability of DP-Fair, IS-DP-Fair and IS-Server, 4 cores, $K = 2$

(d) Schedulability of DP-Fair, IS-DP-Fair and IS-Server, 8 cores, $K = 2$

Figure 5. Non-mixed-critical Isolation Scheduling: Fraction of schedulable task sets vs. system utilization. Red lines in plots correspond to DP-Fair.

- Periods are randomly chosen from $\{x \in \mathbb{Z} \mid 2 \leq x \leq 2000\}$.
- Task utilizations are uniformly chosen from $[0.02, 0.2]$.
- Tasks are equally likely to belong to task class $\{S_1, \cdots, S_K\}$.

For creating dual-criticality, implicit-deadline task sets, we implement a widely used task set generator [6, 16, 26], with the following parameters:

- Probability of any task being HI criticality $P_{\text{HI}} = 0.2$.
- $r = C_i(\text{HI})/C_i(\text{LO})$; for each HI criticality task r is chosen uniformly from $[1, 5]$.
- The utilization of any dual-criticality task set is defined as $\max\{U_{\text{HI}}^{\text{LO}} + U_{\text{LO}}^{\text{LO}}, U_{\text{HI}}^{\text{HI}}\}$.

Periods and LO level utilizations for dual criticality task sets are generated similar to the non mixed-critical task sets. For both basic IS task sets and dual criticality task sets, we perform experiments with system utilizations varying in the interval $[0.1, 4]$ (quad-core experiments) or $[0.1, 8]$ (octa-core experiments). Utilization is incremented in steps of 0.1.

B. Schedulability

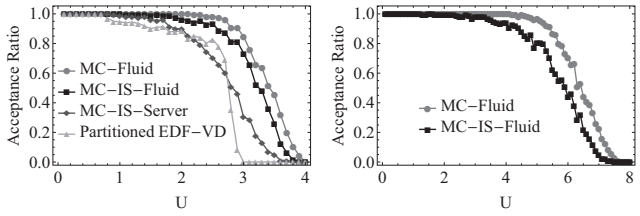
1) Non Mixed-Criticality Isolation Scheduling: First, we evaluate the schedulability loss caused by enforcing the IS constraint (i.e., mutual exclusion among task classes). For this purpose, we generate 500 task sets for each system utilization and compute the fraction of task sets that are deemed schedulable under the IS-DP-Fair approach on $m = 4$ and $m = 8$ cores. The results are depicted in Figure 5a (4 cores) and Figure 5b (8 cores). For both configurations, the acceptance ratio by IS-DP-Fair decreases as the number of classes increases. This is intuitive: with only one task class, IS-DP-Fair is equivalent to DP-Fair and hence optimal. By adding more classes, we limit task parallelism within each class, thus impairing schedulability. Our results here also match with the theoretical analysis. According to Theorem 5, with increasing number of classes or processors, the speedup bound of IS-DP-Fair to catch DP-Fair increases, i.e., schedulability decreases.

Second, we compare schedulability under three different approaches: DP-Fair, IS-DP-Fair and IS-Server. We restrict the number of task classes to two. Our results are presented in Figure 5c (4 cores) and Figure 5d (8 cores). Note that IS-DP-Fair performs closely to DP-Fair: only at high system utilizations ($U > 3.6$ for 4 cores, $U > 7.3$ for 8 cores) there is schedulability loss. For both configurations, however, there is a considerable schedulability loss by adopting the server-based approach, reaching up to 68.6% for 4 cores ($U = 3.9$) and 86.6% for 8 cores ($U = 7.4$). We mainly attribute this loss to the static task partitioning and the time-triggered global server scheduling. In fact, this cost in schedulability for the server approach can be compensated by significantly less synchronous switches between task classes and zero task migrations, as opposed to IS-DP-Fair. A comparison based on this criterion is presented later.

2) Mixed-Criticality Isolation Scheduling: We now present results for dual-criticality systems and compare our approaches to two state-of-the-art scheduling techniques: MC-Fluid [26] and partitioned EDF-VD [8]. We present our results in Figure 6a (4 cores) and Figure 6b (8 cores). As shown in the figures, the feasibility of MC-IS-Fluid is very close to MC-Fluid for all utilizations. Therefore, we conclude that for dual-criticality task systems, the cost of enforcing Isolation Scheduling by MC-IS-Fluid is relatively low. However, as explained for Figure 5a, this cost is expected to increase as the number of criticality levels increase. The server-based approach incurs a loss in schedulability as compared to MC-IS-Fluid, similar to that in the non mixed-criticality scenario. Note, nonetheless, that even with the additional IS constraint, both MC-IS-Fluid and MC-IS-Server are comparable to or better than a well-known mixed-criticality scheduling technique (partitioned EDF-VD) in terms of schedulability. This is a significant result, especially considering that partitioned EDF-VD does not enforce isolation. The reason for this gain is that the optimization formulation for partitioning performs better than the bin-packing heuristic employed by partitioned EDF-VD.

C. Synchronous Task Class Switching & Task Migration

Conceptually, the advantage of adopting the server-based approach over the fluid approach for Isolation Scheduling lies in the reduced number of synchronous task class switches, which can cause a significant runtime overhead.



(a) Schedulability of MC-Fluid, MC- (b) Schedulability of MC-Fluid, MC-IS-Fluid and MC-IS-Server, 4 cores IS-Fluid, 8 cores

Figure 6. Mixed-Criticality Isolation Scheduling.

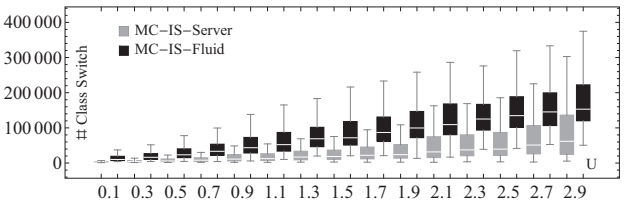


Figure 7. Distribution of task class switches for MC-IS-Fluid and MC-IS-Server for increasing system utilization (box-whisker-plot).

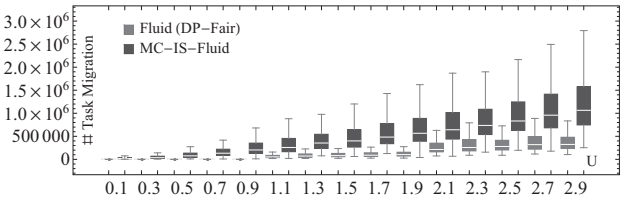


Figure 8. Distribution of task migrations for MC-IS-Fluid and DP-Fair for increasing system utilization (box-whisker-plot).

Additionally, the IS-Server and MC-IS-Server approaches require no task migrations as opposed to the fluid schedulers, since tasks are statically partitioned to cores prior to execution. We now present simulation results to quantify these advantages. Our experiments consider a dual-criticality setting, which is equivalent to supporting $K = 2$ task classes in the non mixed-criticality case. For each utilization point between 0.1 and 4 (in steps of 0.1), we consider 500 randomly generated dual-criticality task sets.

We first compare the required number of synchronous switches between two criticality levels for MC-IS-Fluid and MC-IS-Server. To this end, we consider a fixed, sufficiently large time interval $\Pi = 2 \times 10^6$ time units. The number of class switches within Π for the various task sets is presented for MC-IS-Fluid and MC-IS-Server in Figure 7. Data in these plots are represented in the form of a box-whisker-plot to reveal the distribution of class switches for the 500 considered task sets at each utilization point (median, minimum, maximum). To enhance readability, only the points that lie within the inner fence of the distribution are shown. Also, we consider utilizations up to $U = 3$, since beyond it several task sets are not schedulable under MC-IS-Server (few samples are available). The results confirm the significant reduction of class switches by the MC-IS-Server approach. Given the median of class switches across all utilization points, MC-IS-Server achieves a 2.4 to 4.4-fold reduction in synchronous class switches compared to MC-IS-Fluid.

We present the number of task migrations for MC-IS-Fluid and a non-IS scheduler, namely DP-Fair in Figure 8[2]. Recall that MC-IS-Server incurs no task migration. One can observe that, for low system utilizations, MC-IS-Fluid already incurs task migrations. This is counter-intuitive, as one might expect that task migrations should happen only for system utilizations greater than 1. Although this is true for DP-Fair, it is not true for MC-IS-Fluid. The reason for task migrations at low utilizations for MC-IS-Fluid is two-fold: First, the deadline of HI criticality tasks is shortened in LO mode, effectively increasing system utilization. Second, MC-IS-Fluid adopts IS-DP-Fair scheduling in LO mode, which allocates the smallest possible slices for each criticality level, forcing tasks to migrate in order to increase schedulability. Furthermore, as system utilization increases, both the number of system slices and the migrations within each slice increase[3], as more tasks exist in the system. Across all utilizations, MC-IS-Fluid incurs significantly more task migrations than DP-Fair, e.g., for the median of task migrations and $U \geq 1.1$, the former incurs a 2.8 to 5.4-fold increase as compared to the latter. However, this cost comes with the benefit that MC-IS-Fluid enforces isolation among different task classes and incorporates dynamic scheduling to improve system schedulability.

VIII. Concluding Remarks

We presented the Isolation Scheduling (IS) model, a flexible abstraction that helps designing real-time scheduling policies for multicore processors with shared resources. The IS model leverages the common property of having different *task classes* within a real-time task set and avoids interference among different classes by enforcing by construction the *IS constraint*: at any time, only one task class can run exclusively on the platform. We present four novel scheduling policies built upon the IS model. The goal of the first two policies is enforcing the IS constraint; IS-DP-Fair is based on *fluid* scheduling, while IS-Server is based on *server* scheduling. The remaining policies, MC-IS-Fluid and MC-IS-Server, are variants that target mixed-criticality systems, which we use as a relevant case study to apply the IS model.

Our experimental evaluation indicates that, for typical dual criticality / class systems, enforcing the IS constraint by fluid-based methods incurs relatively small penalty in

[2]For DP-Fair, we take the original deadlines of all tasks since it is a non mixed-criticality scheduler. For the two cases, we simulate task migrations according to Algorithm 2 and [27], respectively.

[3]Maximum $m - 1$ migrations within each slice.

schedulability. Server-based methods have larger theoretical schedulability penalty, but drastically reduce runtime overheads on inter-class switches and task migrations. This advantage would be significant in real-world deployments, where switches and migrations can be costly.

The IS model opens several directions for future research. One direction is to study the trade-off between theoretical schedulability and runtime overhead on real deployments, since high runtime overhead could cause missed deadlines despite satisfaction of the schedulability condition. A second direction is to evaluate alternative partitioning and server scheduling approaches for MC-IS-Server, in order to reduce the schedulability gap compared to MC-IS-Fluid. In general, we hope that the IS model will prove a valuable tool for devising efficient scheduling policies for multicores.

REFERENCES

[1] "RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification," 2011.

[2] "ISO 26262, Road Vehicles - Functional Safety," 2011.

[3] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS*, 1998, pp. 4–13.

[4] J. Anderson, S. Baruah, and B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[5] ARINC, "ARINC 653-1 avionics application software standard interface," http://www.arinc.com/, Tech. Rep., 2003.

[6] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *ECRTS*, 2012, pp. 145–154.

[7] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *STOC*. ACM, 1993, pp. 345–354.

[8] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[9] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS*, 1990, pp. 182–190.

[10] E. Bini, M. Bertogna, and S. Baruah, "Virtual multiprocessor platforms: Specification and use," in *RTSS*, 2009, pp. 437–446.

[11] A. Burns, T. Fleming, and S. Baruah, "Cyclic executives, multi-core platforms and mixed criticality applications," in *ECRTS*, 2015, pp. 3–12.

[12] A. Burns and R. Davis, "Mixed criticality systems-a review," 2015.

[13] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011, pp. 365–376.

[14] J. Flodin, K. Lampka, and W. Yi, "Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks," in *SIES*, 2014, pp. 151–159.

[15] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in *EMSOFT*, 2012, pp. 63–72.

[16] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *EMSOFT*, 2013, pp. 1–15.

[17] ——, "Mapping mixed-criticality applications on multi-core architectures," in *DATE*, 2014, pp. 1–6.

[18] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, pp. 1–51, 2015.

[19] J. Goossens and V. Berten, "Gang ftp scheduling of periodic and parallel rigid real-time tasks," *arXiv preprint arXiv:1006.2617*, 2010.

[20] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *DATE*, 2013, pp. 525–530.

[21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded Software*, ser. LNCS, 2001, vol. 2211, pp. 166–184.

[22] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele, "Service adaptions for mixed-criticality systems," in *ASP-DAC*, 2014, pp. 125 – 130.

[23] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele, "An isolation scheduling model for multicores," ETH Zurich, Laboratory TIK, Tech. Rep. 361, December 2015.

[24] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *RTSS*, 2009, pp. 459–468.

[25] 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, pp. 321–331.

[26] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "Mc-fluid: Fluid model-based mixed-criticality scheduling on multiprocessors," in *RTSS*, 2014, pp. 41–52.

[27] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *ECRTS*, 2010, pp. 3–13.

[28] M. Mollison, J. Erickson, J. Anderson, S. Baruah, J. Scoredos *et al.*, "Mixed-criticality real-time scheduling for multicore systems," in *ICCIT*, 2010, pp. 1864–1871.

[29] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore systems," in *ISCA*, 2009, pp. 57–68.

[30] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: bank privatization for predictability and temporal isolation," in *CODES+ISSS*, 2011, pp. 99–108.

[31] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp," in *ECRTS*, 2012, pp. 155–165.

[32] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale *et al.*, "Single core equivalent virtual machines for hard real-time computing on multicore processors," Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep., November 2014.

[33] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *RTSS*, 2003, pp. 2–13.

[34] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *ECRTS*, 2008, pp. 181–190.

[35] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer, "Idamc: A noc for mixed criticality systems," in *RTCSA*, 2013, pp. 149–156.

[36] D. Tămaş-Selicean and P. Pop, "Design optimization of mixed-criticality real-time embedded systems," *ACM Trans. on Embeddded Computing Systems*, vol. 14, no. 3, pp. 50:1–50:29, 2015.

[37] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007, pp. 239–243.

[38] E. Wandeler and L. Thiele, "Optimal tdma time slot and cycle length allocation for hard real-time systems," in *ASP-DAC*, 2006.

[39] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *RTSS*, 2013, pp. 372–383.

[40] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *ECRTS*, 2012, pp. 299–308.

[41] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *RTAS*, 2014, pp. 155–166.

[42] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 236–247, 2003.