Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
{giannopoulou, stoimenov, phuang, thiele}@tik.ee.ethz.ch

ABSTRACT

A common trend in real-time safety-critical embedded systems is to integrate multiple applications on a single platform. Such systems are known as mixed-criticality (MC) systems as the applications are usually characterized by different criticality levels (CLs). Nowadays, multicore platforms are promoted due to cost and performance benefits. However, certification of multicore MC systems is challenging because concurrently executed applications with different CLs may block each other when accessing shared platform resources. Most of the existing research on multicore MC scheduling ignores the effects of resource sharing on the execution times of applications. This paper proposes a MC scheduling strategy which explicitly accounts for these effects. Applications are executed by a flexible time-triggered criticality-monotonic scheduling scheme. Schedulers on different cores are dynamically synchronized such that only a statically known subset of applications of the same CL can interfere on shared resources, e. g., memories, buses. Therefore, the timing effects of resource sharing are bounded and we quantify them at design time. We combine this scheduling strategy with a mapping optimization technique for achieving better resource utilization. The efficiency of the approach is demonstrated through extensive simulations as well as comparisons with traditional temporal partitioning and state-of-the-art scheduling algorithms. It is also validated on a real-world avionics system.

1. INTRODUCTION

As a result of the prevalence and maturity of multicore systems in the electronics market, the field of embedded systems experiences nowadays an unprecedented trend towards integrating multiple applications into a single platform. This trend applies even in real-time embedded systems for safety-critical domains, such as avionics and automotive. The applications in these domains, however, are usually characterized by several criticality levels, known as *Safety Integrity Levels* (SIL) or *Design Assurance Levels* (DAL). These levels express the required protection against failure when designing a safety-critical system and hence, influence all steps of the specification, design, development, testing, and certification processes.

For the integration of *mixed-criticality* applications on a common platform, the existing certification standards require complete timing and spatial *isolation* among applications of different criticalities so that no interference among them is possible. To achieve isolation on every core, system designers usually rely on partitioning mechanisms at platform level, such as the ones specified by the ARINC-653 standard [4]. What is not trivial is how isolation is achieved when several cores share platform resources, e. g., caches or memory buses, which is a common practice for efficiency and cost reasons.

Obviously, if several cores access e. g., a memory bus in an uncontrolled manner, interference among applications of different criticalities cannot be avoided. Then, a lower criticality application accessing the memory bus can block the access of any other concurrently executed application. If accesses are *synchronous*, causing execution on a core to stall until they are completed (such as memory accesses due to cache misses), then each access of the lower criticality application can affect the response time of higher criticality applications on other cores. This effect is not only existent but it also cannot be quantified, since generally the certification authority (CA) which examines the higher criticality applications does not have any information on the behavior of the lower criticality applications, which are co-hosted in the integrated platform, unless they are certified at the higher level.

To enforce isolation among cores with a shared memory, one could select a statically scheduled memory bus or implement a server on the bus arbiter with a given accessing budget for each core. The effect of resource sharing on the response time of higher criticality applications can be then quantified, since inter-core interference is limited by construction. These solutions come with certain drawbacks. The first one offers no flexibility. The bus schedule cannot change at runtime if more applications need to be considered or the accessing time reserved for an application is not used. The second solution, although potentially more flexible, has a high design and implementation overhead. Also, both solutions are not applicable on multicores with commercially-off-the-shelf components (COTS), where configuration/virtualization of the memory bus by the application designer is not allowed.

In this work, we suggest an alternative solution for timing isolation in mixed-criticality resource-sharing systems. Applications of any criticality can be mapped on all cores and resource accessing is organised such that only a statically known set of applications of the same criticality can interfere at any time instant. Timing isolation on the core level is achieved through a time-triggered scheduling strategy and on the global level (shared resource) through dynamic inter-core synchronisation with a *barrier* mechanism. The points of inter-core synchronisation are defined by the scheduling strategy and vary in runtime to reflect the dynamic behavior of the applications. The suggested solution does not deviate from the basic principles of partitioning defined by the ARINC-653 standard. Timing isolation is preserved despite resource sharing. At the same time, the flexible definition of "partitions", which are synchronised among the cores and dynamically dimensioned based on the runtime requirements, enables efficient resource utilization. Our solution does not require any special hardware support for eliminating/limiting interference on the shared resources, therefore it can be implemented on COTS platforms.

To the best of our knowledge, the suggested mixed-criticality mapping and scheduling strategy is the first to consider the timing effects of resource contention. Most existing scheduling algorithms neglect this for simplification. That is, the corresponding solutions may not be certifiable with the existing standards, since it is unclear how to guarantee timing isolation. To bound the delays induced by resource contention, we adopt the *superblock* model of execution, which is known for its predictability [24]. Based on it, applications are structured as sequences of access and execution phases with known bounds on accesses and computation times.

The contributions of this paper can be summarized as follows:

- We extend the established mixed-criticality (MC) task model from literature to reflect not only the execution profiles of the tasks at several criticality levels, but also the corresponding memory access profiles.
- We suggest a partitioned scheduling strategy for MC periodic task sets, which (i) ensures timing isolation among different criticality levels (certifiable), (ii) accounts for the effect of memory contention on task execution, (iii) enables efficient resource utilization. The scheduling strategy combines time-triggered and event-driven task activation and can support fixed preemption points.
- We propose a tool for the analysis and design optimization of MC task sets on resource-sharing multicores. The tool takes as input the model of the task set and the platform and generates a mapping of the tasks on cores and a schedule for each core based on a simulated-annealing heuristics approach.
- We perform extensive simulations using synthetic benchmarks and an industrial application to validate the efficiency of our mapping and scheduling method against state-of-the-art methods and its applicability to real-world problems.

2. RELATED WORK

Scheduling of mixed-criticality applications is an emerging research field, which has been attracting increasing attention in recent years. Vestal was the first to introduce the currently dominating MC task model in [30]. He suggested a fixed-task-priority scheduling strategy, which was later [10] proven to be optimal. Baruah et al. [8] extended Vestal's algorithm to assign fixed priorities on job (rather than task) level. In their work, the MC jobs belong to applications that need to be validated by different certification authorities (CA) and they assume that the more critical an application, the more pessimistic the CA will be in the estimation of its WCET. Given the intractability of the MC scheduling problem in this context [6], Baruah et al. proposed in [8] two sufficient schedulability conditions, the worst-case-reservations and own-criticality-based-priority (OCBP) conditions. This work was extended in [19], which introduced a fixed-job-priority scheduling strategy based on the OCBP condition, as well as in several subsequent publications [5, 7, 11, 14]. All of the above works regard scheduling on single cores.

One of the first attempts to extend the MC scheduling strategies (particularly, EDF-VD [5]) to multicore systems was made in [20]. Mollison et al. proposed a scheduling approach for MC tasks on multicores, adopting different strategies (partitioned EDF, global EDF, cyclic executive) for different criticality levels and providing timing isolation through a bandwidth reservation server [3, 22]. Kelly et al. addressed in [15] the problem of *partitioned* fixed-priority preemptive MC scheduling on multicores and assessed empirically alternative solutions for task mapping and priority assignment. Also, Pathan presented in [23] a novel schedulability test for *global* MC fixed-priority scheduling. Finally, following the current industrial practice for certification, which requires strict timing and spatial isolation among tasks of different criticalities, Tamas-Selicean and Pop presented in [29] an optimization method for the mapping and time-triggered scheduling of MC tasks on multicores, complying with the ARINC-653 standard. Most of the above works ignored, though, the inter-core interference on shared platform resources and its effect on schedulability. We claim that this can be dangerous since it has been shown empirically [24] that traffic on the memory bus in COTS-based systems can increase the response time of a real-time task up to 44%. Only [29] considers inter-task communication via message passing, but the message transmission occurs asynchronously over a broadcast time-triggered bus such that no task's execution is blocked. This requires that no shared memory exists and that the bus schedule can be manually configured, assumptions which do not necessarily hold on COTS platforms. Our work considers explicitly the inter-core interference on shared resources, which may be arbitrated according to any (time-driven or event-driven) policy.

The field of worst-case response time (WCRT) analysis under resource contention on multicores is also not new. Several recent works proposed analytic methods to bound the resulting waiting times. Schliecker et al. [26] and Dasari et al. [9] used event models and Schranzhofer et al. [25, 27, 28] arrival curves [18] to model the arrival of access requests from different cores and bound the tasks' WCRT for several event or time-triggered arbitration strategies. These methods relied on over-approximations of the resource arbiter behavior, which can lead to very pessimistic results, esp. for event-driven arbitration. To tackle pessimism, Lv et al. combined abstract interpretation and model checking [21] to represent accurately the resource arbiter. However, due to the state space explosion problem the scalability of their method was restricted for systems with more than 2 cores. Recently, a hybrid analytic/model-checking-based method, combining timed automata [2] for the modeling of the resource arbiter and arrival curves for the access request patterns, was introduced in [13]. The proposed method can be applied to systems with resource arbiters of any complexity (e. g., FCFS, TDMA, FlexRay) and was shown to provide accurate results with satisfying scalability. The last method is applied in our work to analyse the WCRT of the tasks under different mapping/scheduling configurations.

To our knowledge, the only work which has addressed WCRT analysis in MC resource-sharing multicore systems is [32]. The authors proposed a software-based memory throttling mechanism to explicitly control the inter-core interference on the shared memory path. Specifically, assuming that all high criticality tasks are mapped on the same core, lower criticality tasks executing on other cores are assigned a limited memory budget, so that schedulability of the former tasks is guaranteed, while the performance impact on the latter is kept minimal. Our work differs in that (i) tasks of any criticality can be mapped on any core for increased system utilization, (ii) we seek an optimized task mapping and scheduling (it is not given), and (iii) shared memory accessing does not depend on predefined budgets, but on dynamic inter-core synchronisation, which allows for timing isolation among different criticalities.

3. SYSTEM MODEL AND DEFINITIONS

This section defines the task and platform models as well as the requirements that a MC scheduling strategy must fulfil for certifiability. The task models and requirements are based on the established MC assumptions in literature, but also on an avionics case study we addressed for an industrial collaboration. A description of the avionics application (flight management system, FMS) exists in Sec. A.1.

3.1 Task model

We consider mixed-criticality periodic task sets $\tau = \{\tau_1, \dots, \tau_n\}$ with criticality levels (CLs) among 1 (lowest) and L (highest). A task is characterized by a 5-tuple $\tau_i = \{W_i, \chi_i, \mathbf{C}_i, C_{i,deg}, \mathcal{D}ep\}$, where:

- $W_i \in \mathbb{N}^+$ is the period,
- $\chi_i \in \{1, \dots, L\}$ is the criticality level,
- $\mathbf{C}_i$ is a size-L vector of execution profiles, where $C_i(\ell)$ represents an estimation of the computation time and the memory accesses of τ_i at criticality level ℓ, in the form of a hierarchical list (defined later),
- $C_{i,deg}$ is a special execution profile for the cases when τ_i (with $\chi_i < L$) runs in degraded mode. This profile corresponds to the minimum required functionality for τ_i so that no catastrophic effect occurs in the system. If the execution of τ_i can be skipped without catastrophic effects, then this execution profile contains zeros,
- $\mathcal{D}ep(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph representing dependencies among tasks with equal periods. Each node $\tau_i \in \mathcal{V}$ represents a task. An edge $e \in \mathcal{E}$ from τ_i to τ_k implies that within a period the job of τ_i must precede that of τ_k.

For simplicity, we assume that the first job of all tasks is released at time 0 and that the relative deadline D_i of τ_i is equal to its period, i. e., $D_i = W_i$. Furthermore, for the purpose of tight resource interference analysis, we assume that each task follows the dedicated superblock model of execution [27], which has been shown to yield increased timing predictability and is well-suited for the kind of safety-critical real-time applications we are considering [12]. Based on it, each task τ_i is structured as a sequence $\mathcal{S}_i$ of computation phases where only local computation is performed, and access phases where (successive) memory accesses occur. A task may have an arbitrary number of phases and they can be in any order. A computation phase is characterized by a min. and max. execution time and an access phase by a min. and max. number of memory accesses. The j-th phase of task τ_i is denoted as $s_{i,j} \in \mathcal{S}_i$ and is defined by the 4-element list: $\{\mu_{i,j}^{min}(\ell), \mu_{i,j}^{max}(\ell), ex_{i,j}^{min}(\ell), ex_{i,j}^{max}(\ell)\}$, $\forall \ell \in \{1, \dots, L\}$. If $s_{i,j}$ is a computation phase, then $\mu_{i,j}^{min}(\ell) = \mu_{i,j}^{max}(\ell) = 0$. If it is an access phase, then $ex_{i,j}^{min}(\ell) = ex_{i,j}^{max}(\ell) = 0$. The specification of the above parameters for all phases $s_{i,j} \in \mathcal{S}_i$ at CL ℓ yields the level-ℓ execution profile of τ_i, $C_i(\ell)$ (list of $|\mathcal{S}_i|$ lists).

We assume that the worst-case parameters of $C_i(\ell)$ are monotonically increasing for increasing ℓ and the best-case parameters are monotonically decreasing, respectively. This implies that the min./max. interval of execution times or memory accesses for each phase $s_{i,j}$ in $C_i(\ell)$ is included in the corresponding interval of $C_i(\ell + 1)$. Note that the best-case parameters are only needed to obtain more accurate results from the resource interference analysis in Section 4.3.

The different values for the execution times and memory accesses of each profile can be obtained by different tools. For instance, at the lowest level of assurance ($\ell = 1$), the system designer may extract these parameters by profiling and measurement, as in [24]. At higher levels, the CAs may use static analysis tools with more and more conservative assumptions as the required confidence increases. The execution profile $C_i(\ell)$ for each task τ_i is derived only for CLs that are not greater than its own CL, χ_i. For all $\ell > \chi_i$, $C_i(\ell) = C_{i,deg}$. That is because we assume that when certification is done at level of assurance ℓ, all tasks with a lower CL are ignored. However, at runtime, if enough resources are available, the lower criticality tasks can be let to run in degraded mode. Then, the degraded profiles $C_{i,deg}$ are required for scheduling analysis.

3.2 Multicore Resource-Sharing Architecture

We consider a set $\mathcal{P}$ of m processing cores, $\mathcal{P} = \{p_1, \dots, p_m\}$. We assume identical cores but our approach can be easily generalized to heterogeneous platforms. Each core in $\mathcal{P}$ has access to a private local memory and also to a shared (global) memory. Data and instructions are fetched from the shared memory to the local during the access phases of a task, and after each computation phase, the modified data are written back to the shared memory during subsequent access phases. We assume h/w platforms without timing anomalies, such as the fully timing compositional architecture [31], where execution and communication times can be decoupled.

The bus to the global memory is shared among all cores and access to it can be arbitrated according to any event- or time-triggered scheme. We assume that only one core can access the bus at a time and that once granted, a memory access is completed within a fixed time interval, T_{acc} (same for read/write operations). In the meanwhile, pending access requests from other cores stall execution on their cores until they are served.

The shared memory can be single (contiguous) or partitioned, namely split into several memory banks. We assume that the mapping of data/instructions (set of memory blocks $\mathcal{MB}$) to memory banks (set Mem) is known. Particularly, all required information for interference analysis is represented by the *interference graph* $\mathcal{I}(\mathcal{V}_\mathcal{I}, \mathcal{E}_\mathcal{I})$, where $\mathcal{V}_\mathcal{I} = \mathcal{V}_\tau \cup \mathcal{V}_{\mathcal{MB}} \cup \mathcal{V}_{Mem}$. $\mathcal{V}_\tau$ represents all tasks in τ, $\mathcal{V}_{\mathcal{MB}}$ all memory blocks and $\mathcal{V}_{Mem}$ all memory banks in the system. $\mathcal{I}$ is composed by two sub-graphs; (i) the bipartite graph $\mathcal{I}_1(\mathcal{V}_\tau \cup \mathcal{V}_{\mathcal{MB}}, \mathcal{E}_1)$, where an edge from $\tau_i \in \mathcal{V}_\tau$ to $mb_j \in \mathcal{V}_{\mathcal{MB}}$ implies that task τ_i reads or writes from/on memory block mb_j, and (ii) the bipartite graph $\mathcal{I}_2(\mathcal{V}_{\mathcal{MB}} \cup \mathcal{V}_{Mem}, \mathcal{E}_2)$, where an edge from $mb_j \in \mathcal{V}_{\mathcal{MB}}$ to $mem_k \in \mathcal{V}_{Mem}$ denotes the allocation of memory block mb_j on memory bank mem_k.

Definition 1. Tasks τ_i and τ_j are *interfering* if and only if $\exists k, l, r \in \mathbb{N}^+ : (\tau_i, mb_k) \in \mathcal{E}_1$, $(\tau_j, mb_l) \in \mathcal{E}_1$ and $(mb_k, mem_r) \in \mathcal{E}_2$, $(mb_l, mem_r) \in \mathcal{E}_2$. $\square$

Figure 1 presents sample interferences graphs for a set of 4 tasks and 3 memory blocks, when the latter are allocated on a single or a partitioned shared memory. Square, ellipsoid and diamond nodes denote respectively, tasks, memory blocks, and memory banks. Note that in Figure 1(a) all tasks are interfering. In Figure 1(b) tasks τ_1 and τ_2 on one hand,

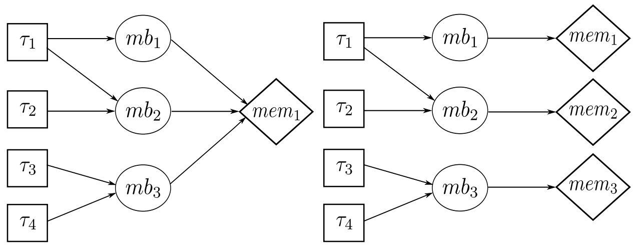


Figure 1: Interference graph $\mathcal{I}$ for (a) single shared memory, (b) partitioned shared memory (3 banks)

and τ_3, τ_4 on the other hand, are interfering. The interfering tasks can delay each other when executed in parallel.

The mapping of tasks on the platform is defined by function $\mathcal{M} : \tau \to \mathcal{P}$. If any mapping constraints exist, they are represented by a bipartite graph $\mathcal{M}_{con}(\mathcal{V}_\tau \cup \mathcal{V}_\mathcal{P}, \mathcal{E})$, where an edge from $\tau_i \in \mathcal{V}_\tau$ to $p_j \in \mathcal{V}_\mathcal{P}$ implies that task τ_i cannot be mapped on core p_j. Note that $\mathcal{M}$ is not given, but it will be determined by our approach.

3.3 MC Scheduling Requirements

Under the above system assumptions, we seek a *correct* scheduling strategy for the MC task set τ on $\mathcal{P}$, which will enable *composable* and *incremental certifiability.* We define the properties of correctness, composable and incremental certifiability, which are crucial for a successful and economical certification process, below.

Definition 2. A scheduling strategy is *correct* if it schedules any task set τ such that the provided schedule is admissible at all criticality levels. A schedule of τ is *admissible* at CL ℓ if and only if:

- the jobs of each task τ_i, satisfying $\chi_i \geq \ell$, receive enough resources between their release time and deadline to meet their real-time requirements according to execution profile $C_i(\ell)$,
- the jobs of each task τ_i, satisfying $\chi_i < \ell$, receive enough resources between their release time and deadline to meet their real-time requirements according to execution profile $C_{i,deg}$. $\square$

The term *resources*, in this context, refers to both processing time and access to the shared memory.

Definition 3. A scheduling strategy enables *composable certifiability* if all tasks of a CL ℓ are temporally isolated from tasks with lower CLs, for all $\ell \in \{1, \dots, L\}$. Namely, the execution and access activities of a task τ_i must not delay in any way any task with CL greater than χ_i. $\square$

The requirement for composability enables different CAs to certify task subsets τ_ℓ of a particular CL ($\tau_\ell \subseteq \tau$) even without any knowledge of the tasks with lower CLs in τ. This is important when several CAs need to certify not the whole system, but individual parts of it. Each CA still needs some information on the scheduling of tasks with higher CL than the one considered. Such information can be provided by the responsible CAs for the higher-criticality task subsets.

Definition 4. A scheduling strategy enables *incremental certifiability* if the real-time properties of the tasks at all criticality levels $\ell \in \{1, \dots, L\}$ are preserved when new tasks are added to the system. $\square$

This property implies that if the schedule of a task set τ is certified as admissible, the certification process will not need to be repeated if new tasks are added later to the system.

4. FLEXIBLE TIME-TRIGGERED IMPLEMENTATION

The problem that we are addressing can be formulated as follows. Given: (i) a periodic MC task set τ, (ii) an architecture consisting of processing cores $\mathcal{P}$ that share a (single or partitioned) memory, (iii) the memory interference graph $\mathcal{I}$ and (iv) mapping constraints $\mathcal{M}_{con}$, determine: (i) the mapping $\mathcal{M}$ of tasks to cores and (ii) the schedule on each core, such that all tasks meet their MC real-time requirements at all levels of assurance and the workload is balanced among the cores.

First, we consider a scheduling strategy with a given mapping. The strategy combines time and event-triggered task activation to impose timing isolation among different CLs. It is non-preemptive, but it supports fixed preemption points. Next, we propose a heuristic optimization method for finding a mapping and a schedule on a resource-sharing multicore platform.

4.1 Scheduling

In a resource-sharing multicore system, where inter-core interference on a shared resource cannot be avoided/limited, a way to impose timing isolation among different CLs is to allow only tasks of the same CL to be executed at a time. Hence, we suggest scheduling the periodic jobs of τ using a Time-Triggered and Synchronisation-based strategy, denoted as *global TTS* schedule. We assume that on the target platform, it is possible to achieve global clock synchronization among cores.

A global TTS schedule repeats over a *scheduling cycle* with a period equal to the hyper-period H of the tasks in τ, i.e., the least common multiple of their periods. The scheduling cycle consists of *frames* (set $\mathcal{F}$), which start and finish synchronously on all cores. The frame lengths are fixed and they can differ. Their maximum length, however, is restricted by the minimum period in τ. Additionally, each frame is divided into so many *sub-frames* as the number of criticality levels L in the system.

The beginning of a TTS sub-frame is not time-triggered, but achieved through inter-core synchronisation with a barrier mechanism, for the sake of efficient resource utilization. Specifically, a new TTS sub-frame starts once all tasks that were scheduled in the previous sub-frame complete execution across all cores (exception: the first sub-frame of each frame starts upon frame start). The tasks that are scheduled within a sub-frame are always defined by the same CL and the sub-frames within a frame correspond to CLs of decreasing order, i. e., the first sub-frame includes tasks of the highest CL and the last sub-frame includes tasks of the lowest CL. Within a sub-frame, task scheduling on each core is sequential, following a predefined order, namely every task is triggered upon completion of the previous one.

An illustration of a global TTS schedule is given in Figure 2. The parameters of the task set τ are shown in Table 1 (execution times in ms) and we assume no task dependencies. The TTS schedule has a cycle of $H = lcm(100, 50, 50, 200) = 200$ ms and is divided into 4 frames of equal length (50 ms), each with 2 sub-frames ($L = 2$). The static schedule on each core includes $\frac{H}{W_i}$ invocations of each mapped task τ_i, equal to the number of jobs of τ_i that arrive within a hyper-period. The solid lines define the frames and the dashed lines the sub-frames, i.e., potential points, where barrier synchronisation is performed.

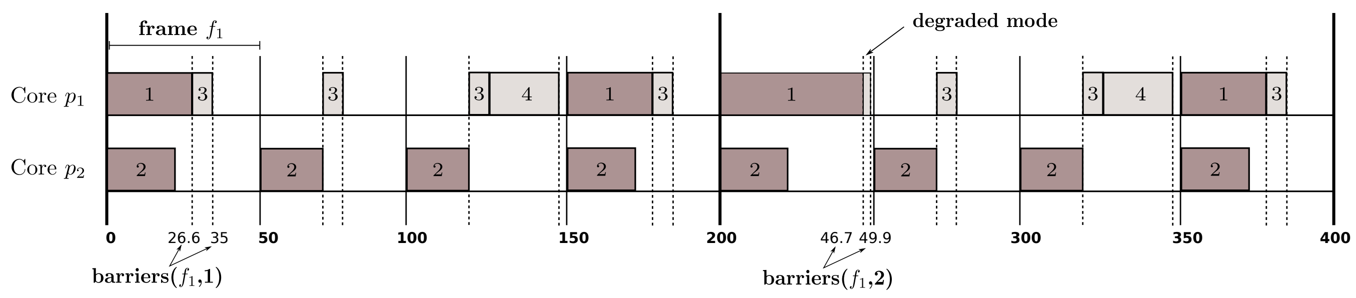


Figure 2: Global TTS Schedule for 2 cycles with task parameters from Table 1 (dark: CL 2, light: CL 1)

At runtime, the length of each sub-frame varies based on the different computation times and accessing patterns that the tasks exhibit. However, its worst-case length can be computed offline for each schedule and each CL using interference analysis (Sec. 4.3). We use function *barriers* : $\mathcal{F} \times \{1, \ldots, L\} \rightarrow \mathbb{R}^L$ to denote the worst-case length of all sub-frames in a particular frame, at a level of assurance. In Figure 2, vector $barriers(f_1, 1)$ (1st cycle) indicates the worst-case lengths of the first frame's sub-frames for the level-1 task execution profiles. Respectively, $barriers(f_1, 2)$ (2nd cycle) indicates the worst-case lengths of the same sub-frames when the execution profiles $C_1(2), C_2(2), C_3(2)$ are considered. We use $barriers(f, \ell)_i$ to denote the worst-case length of the i-th subframe of f at CL ℓ.

The dynamic barriers are used to achieve flexibility and efficient resource utilization. That is, if sub-frames started at fixed time points, they would have to be dimensioned for the worst-case execution profiles of the tasks, so that all possible execution scenarios are covered. This is a common practice, e.g., when dimensioning the timing partitions in ARINC-653 architectures, where only the execution profile at each task's own CL is considered [29]. This approach can be very inefficient since the higher criticality tasks may never reveal the corresponding execution profiles in practice. Resources, however, are reserved for them, leading to large slack times, during which cores are idle, but cannot be used by tasks of lower CL. The barrier synchronisation, on the other hand, occurs dynamically depending on the execution scenarios revealed at runtime, thus enabling efficient resource utilization.

Runtime behavior. Given an admissible schedule S and the *barriers* function, the local scheduler on each core manages task execution within each frame $f \in \mathcal{F}$ as follows:

- For the i-th sub-frame, the scheduler triggers sequentially the corresponding jobs based on the schedule table of S. Upon completion of the jobs' execution, it signals the event and waits until the barrier synchronisation of all cores.
- Let the elapsed time from the beginning of the i-th sub-frame until the barrier synchronisation be t. Given the CL ℓ which satisfies the inequality:

$$t \leq \min_{\ell \in \{1,\ldots,L\}} \{barriers(f, \ell)_i\}, \tag{1}$$

  the scheduler will trigger jobs in the next sub-frame such that tasks with CL lower than ℓ run in degraded mode. If $\ell = 1$, then jobs in the next sub-frame will be triggered in normal mode.
- The two previous steps are repeated for each sub-frame, until the next frame is reached.

Note that the decision on whether a task will run in degraded mode regards only the current frame and is not relevant for the subsequent frames. An exceptional case arises when barrier synchronisation in a sub-frame is not achieved by the time indicated by $barriers(f, L)$. Since schedule S is admissible, this case should never occur and hence, it indicates *erroneous behavior* of the system.

Table 1: Task set definition

τ_i	χ_i	W_i	$\mathbf{C}_i$	$C_{i,deg}$
τ_1	2	100	$C_1(1) = \{\{10, 14, 0, 0\}, \{0, 0, 20, 25\}, \{6, 8, 0, 0\}\}$ $C_1(2) = \{\{8, 30, 0, 0\}, \{0, 0, 15, 44\}, \{6, 12, 0, 0\}\}$	N/A
τ_2	2	50	$C_2(1) = \{\{8, 10, 0, 0\}, \{0, 0, 15, 18\}, \{1, 2, 0, 0\}\}$ $C_2(2) = \{\{5, 12, 0, 0\}, \{0, 0, 15, 20\}, \{1, 4, 0, 0\}\}$	N/A
τ_3	1	50	$C_3(1) = \{\{4, 5, 0, 0\}, \{0, 0, 6, 8\}, \{2, 4, 0, 0\}\}$ $C_3(2) = C_{3,deg}$	$C_{3,deg} = \{\{2, 2, 0, 0\}, \{0, 0, 2, 3\}, \{1, 2, 0, 0\}\}$
τ_4	1	200	$C_4(1) = \{\{10, 10, 0, 0\}, \{0, 0, 20, 20\}, \{10, 10, 0, 0\}\}$ $C_4(2) = C_{4,deg}$	$C_{4,deg} = \{\{0, 0, 0, 0\}, \{0, 0, 0, 0\}, \{0, 0, 0, 0\}\}$

Admissibility. Let S be a TTS schedule constructed such that all jobs (in total $\frac{H}{W_i}$ jobs) of every task $\tau_i \in \tau$ are scheduled in frames within their release time and deadlines and all dependency and mapping constraints hold, as in the case of Figure 2. To evaluate the admissibility of S, one needs to compute function *barriers* for each frame $f \in \mathcal{F}$ and each level of assurance $\ell \in \{1, \ldots, L\}$. The derivation of $barriers(f, \ell)$ is discussed in detail in Sec. 4.3. S is ℓ-admissible if and only if it fulfils the following condition:

$$\sum_{i=1}^{L} barriers(f, \ell)_i \leq \mathcal{L}_f, \forall f \in \mathcal{F}, \tag{2}$$

where $\mathcal{L}_f$ denotes the length of frame f. If the condition holds for all frames $f \in \mathcal{F}$, it follows that all scheduled jobs in S can meet their deadlines at level of assurance ℓ. If it holds also for all CLs, then schedule S is admissible according to Definition 2 of Sec. 3.3. Hence, it can be accepted by any CA at any level of assurance and the scheduling strategy is correct.

Recall that if different CAs certify task subsets of different CLs, then for composable certifiability (Definition 3), the CAs of lower-criticality subsets need some information concerning the resource allocation for the higher-criticality subsets. For the TTS scheduling strategy, this information is fully represented by function *barriers*. Therefore, global TTS enables composable certifiability. Similarly, it enables incremental certifiability (Definition 4), since new tasks can be inserted into their respective CL sub-frame if there is sufficient slack time in the frame.

Benefits and Challenges. The global TTS scheduling strategy features certain advantages w. r. t. its applicability and certifiability. It is easy to implement even on COTS architectures, since contention on the shared memory does not need to be eliminated by hardware. It enforces timing isolation among different CLs and enables composable and incremental certifiability. It enables efficient resource utilization, since the sub-frames are dynamically adapted to the runtime task execution profiles. It can force tasks to execute in degraded mode if a higher-criticality task exhibits a level-ℓ execution profile with $\ell > 1$, but only for the duration of a TTS frame and not for the rest of the system's life-time, as in previous MC scheduling strategies, e. g., [7]. Finally, it can handle erroneous behavior, by aborting the responsible task execution and continuing scheduling as normal from the beginning of the next frame.

These advantages come at a cost, namely the runtime overhead for clock and barrier synchronisation among the cores. However, several mechanisms (hardware or software, centralized or distributed) exist to achieve this synchronisation. The strategy comes also with certain limitations, e. g., the time-triggered frames and the fixed preemption points. Their impact on schedulability is evaluated in Sec. 5.

4.2 Mapping Optimization

The problem of optimal task mapping on multiple cores is known to be NP-hard, resembling the combinatorial bin-packing problem. To tackle this challenge, we propose the *MC Mapping and Scheduling Optimization* (MCMSO) method. MCMSO takes as input the periodic task set τ, the set of cores $\mathcal{P}$ along with the interference graph $\mathcal{I}$ and the mapping constraints $\mathcal{M}_{con}$, and returns the mapping function $\mathcal{M}$ of tasks to cores and an admissible schedule S if there exists one. S consists of the dimensions of the TTS cycle (period H and lengths of frames $\mathcal{F}$) and a static schedule table for each core in $\mathcal{P}$. The schedule table defines the sequence of task execution in each frame and indicates the tasks, after which barrier synchronisation is performed.

MCMSO implements a heuristic method based on simulated annealing (SA) [16] (see Sec. A.2). SA is only one of the methods that can be applied for the design optimization. If the optimization problem itself was the focus, one could consider also e. g., constraint solvers or other non black box heuristics. This is, however, outside the scope of this paper. In summary, the MCMSO approach is described by the following steps:

1. Dimension the TTS scheduling cycle and frame lengths based on the periods of tasks in τ[1].
2. Generate a random mapping/schedule of the jobs of τ within H on the cores of $\mathcal{P}$ and the frames $\mathcal{F}$ of the TTS cycle, such that all constraints are respected.
3. Apply a simulated annealing approach to generate and explore neighboring mappings (assignments of tasks to cores) and schedules (assignment of jobs to frames), until an optimal solution is found or a given computational budget is exhausted. A mapping/scheduling solution is considered optimal if all jobs meet their deadlines at all levels of assurance (admissible) and the worst-case sub-frame lengths are minimized.

Optimization heuristics. The SA algorithm seeks the global minimum of a given cost function in a state space. It begins with an arbitrary solution (state) and it considers a series of random transitions based on a neighbourhood function. At each step, if the neighbouring state S' is of lower cost than the current state S, SA accepts the transition to S'. Else, SA accepts the transition with probability $e^{-\left(Cost(S')-Cost(S)\right)/T}$, where T is a positive constant, commonly known as *temperature*.

[1]The lengths of the frames can be also optimized by MCMSO.

In this work, the cost function of a candidate solution S is defined as:

$$Cost(S) = \begin{cases} c_1 = \max_{f \in \mathcal{F}} \left\{ \max_{\ell \in \{1,\dots,L\}} late(f,\ell) \right\} & \text{if } c_1 > 0 \\ c_2 = \|barriers\|_3 & \text{if } c_1 \leq 0 \end{cases}$$

where $late(f,\ell)$ expresses the difference between the worst-case completion time of the last sub-frame of f and the length of f:

$$late(f,\ell) = \sum_{i=1}^{L} barriers(f,\ell)_i - \mathcal{L}_f. \tag{3}$$

If $late(f,\ell) > 0$, the tasks in f cannot complete execution by the end of the frame for their ℓ-level execution profiles. Note that the *barriers* function is computed for each visited state using a fast and conservative interference analysis method as explained in Section 4.3.

With this cost function, we initially guide the design space exploration to find an admissible solution. If at least one task cannot complete execution within its frame, at any level of assurance ℓ, the term $late(f,\ell)$ will be positive and so will be c_1. Otherwise, c_1 will be negative or 0, implying that all tasks "fit" into the corresponding frames. In this case, c_2, the 3rd norm of all sub-frame lengths, $\forall f \in \mathcal{F}, \forall \ell \in \{1,\dots,L\}$, is used as the cost function to minimize the worst-case lengths of all sub-frames for the admissible solution. Minimizing the sub-frame lengths is important for a balanced workload distribution and also, for incremental certifiability (maximization of slack times). Note that depending on how slack time can be used for future tasks, other optimization criteria, e. g., the number of empty frames, can also be considered in the cost function.

Design space exploration is restricted only to possible solutions, which fulfil the following criteria:

- a total of $\frac{H}{W_i}$ jobs of each task τ_i are scheduled in the TTS cycle, in frames within their respective release times and deadlines,
- all jobs of a task τ_i are scheduled on the same core, respecting constraints $\mathcal{M}_{con}$,
- all jobs of a task τ_k, which depends on τ_i, are scheduled on the same core as τ_i. Each job of τ_k is scheduled in the same or a later frame than the corresponding job of τ_i within their common period. If they are scheduled in the same frame, the job of τ_k must succeed that of τ_i.

If no candidate solution can be found to satisfy the above criteria, the search is aborted, i. e., τ is considered not TTS-schedulable on $\mathcal{P}$.

The neighboring function is defined by two possible transitions. Namely, a new solution S' is generated by selecting randomly a task of τ and either (i) re-mapping all its jobs (and all of its dependent tasks' jobs) on another core (*core variation*) or (ii) re-scheduling one of its jobs to another TTS frame on the same core (*frame variation*). In both cases, the alternative core or frame are selected non-deterministically. The probability of choosing between the two variations is given as input to the SA algorithm.

Extension: Preemption Points. In certain cases (see the avionics FMS case, Sec. A.1), some sort of preemption is indispensable for schedulability, esp. when one considers task sets with one or more computationally intensive tasks, which may not "fit" in any frame of a global TTS schedule. Enabling a preemptive scheduling strategy, where each task can start executing in one sub-frame and continue over several frames (always in the corresponding sub-frame of its CL) would not be efficient because:

- The computationally intensive tasks (in the FMS, they are of the highest CL) would be allowed to run up to the end of each frame, in which they are scheduled, thus preventing any other tasks of the same or lower CL from being executed. This behavior would affect not only tasks on the same core, but also on the remaining cores, since the barrier synchronisation for the corresponding sub-frame would not be achieved.
- The fact that these tasks could be preempted at any possible point of execution makes accurate interference analysis impossible, since the execution profiles of the tasks (including memory accesses) cannot be extracted for any possible partial execution.

To tackle this challenge, we use the concept of *fixed preemption points*. A task may have a certain amount of well-defined preemption points, so that execution profiles for the corresponding partial executions can be extracted. We specify each "preemptable" task by a list of alternative executions, e. g., one with no preemption, one with 1 preemption point, etc. In each case, the partial executions are defined as chains of dependent tasks with the same period, e. g., a chain with one task or 2 dependent tasks, respectively. The alternatives are given as input to MCMSO. An admissible solution S must eventually include one of the alternative executions of the related tasks. Note that this extension introduces a new transition for the neighboring function. Namely a new solution S' can be generated also by selecting randomly a task chain and substituting it for one of its alternative executions (*alternative variation*).

Extension: Top-k Solution Ranking. MCMSO can be extended to return not the best encountered solution w. r. t. the cost function, but the top k solutions, with $k > 1$. For this purpose, it maintains a list S_{best} of length k. S_{best} is updated every time a new solution is visited if the new solution is better than at least one entry in the list. Solution ranking can be useful, e. g., if no admissible schedule can be found. In this case, the top-k solutions can be further examined, i. e., tighter interference analysis can be applied to confirm that they are indeed not admissible (explained in Sec. 4.3) or their structure can be examined to reveal the reason of schedulability failure (e. g., need for preemption for long tasks). Note that a top solution found with a conservative interference analysis may not correspond to the top solution found by using tighter interference analysis. However, if a non-admissible top solution is found with a conservative method, we can consider this as a good initial solution for a tighter interference analysis.

4.3 Interference Analysis

WCRT analysis for multicores under resource contention scenarios is a highly complex problem. In this paper, we consider synchronous memory accesses. Therefore, the delays incurred every time a core is waiting to access the shared memory must be explicitly considered for its WCRT analysis. Providing tight bounds for these delays is far from trivial because one needs to consider (i) the pending access requests from all other cores and (ii) the state of the memory arbiter. The number of possible interleavings of accesses from different cores can be very large. Recall that for the superblock model, the access phases are defined by a min./max. range of potential accesses and also, the starting times of these phases are highly variable, depending on the actual duration of the preceding access and computation phases. Moreover, knowing the state of the arbiter over time is challenging, esp. when event-driven arbitration policies are considered.

For this problem, we apply the WCRT analysis methodology from [13]. This suggests state-based modeling and analysis of the multicore system, using timed automata (TA) to model accurately memory arbiters of any complexity and model-checking to estimate the WCRT of a task. The resource accesses from different cores can be modeled using either TA (all possible scenarios) or arrival curves [18] (over-approximation). An arrival curve represents abstractly the maximum number of accesses that a core can issue in any time interval and can be also incorporated into the TA system model [17]. Deciding between the two alternatives for the memory access representation yields a trade-off between accuracy (exploration of all possible access interleavings) and scalability (reduction of state space).

The selected method needs few seconds up to several minutes for the WCRT estimation of a task, depending on the problem size. However, during design space exploration (DSE) with MCMSO, usually thousands of solutions are explored, so a fast analysis methodology must be favored. Thus, in MCMSO we apply a conservative method to derive the required worst-case sub-frame lengths. E.g., if the shared memory is FCFS-arbitrated, we assume that all accesses of a task are delayed by all cores with at least one interfering task (based on $\mathcal{I}$) scheduled in the same sub-frame. This is a pessimistic assumption, nevertheless it leads to safe WCRT bounds, computed as follows:

$$WCRT_{i,cons}(\ell) = \sum_{s_{i,j} \in \mathcal{S}_i} ex_{i,j}^{max}(\ell) + m_{int} \cdot \mu_{i,j}^{max}(\ell) \cdot T_{acc} \quad (4)$$

where m_{int} is the number of cores with at least one concurrently running interfering task. At the end of the DSE phase, in order to refine the information conveyed by function *barriers*, we apply the method of [13] to the best found solution. The result of this analysis can be used then for the certification procedure. If no admissible solution is found, the same method can be applied to the k top encountered solutions ($k \geq 1$), as this step may reveal admissible schedules.

Consider now the schedule of Figure 2, where we apply the two methods to derive $barriers(f_1, 1)$ and $barriers(f_1, 2)$. Let the interference graph $\mathcal{I}$ for the target dual-core architecture be one of Figure 1. In both cases, τ_1 and τ_2 are interfering tasks. Let the memory access latency be $T_{acc} = 0.05$ ms. Applying Eq. 4 for τ_1 and τ_2 at $\ell = 1$ yields: $WCRT_{1,cons}(1) = 27.2$ ms, $WCRT_{2,cons}(1) = 19.2$ ms. So, the worst-case length of the 1st sub-frame in f_1 is 27.2 ms for the level-1 execution profiles of the tasks. This way, we derive the *barrier* vectors of Table 2. The same table shows the corresponding results with the method of [13] (accesses modeled by TA). Note that the refinement of *barriers* with the second method satisfies the criterion of Eq. 2 for frame f_1 ($46.7 + 3.2 < 50$), which was not true for the results of the first method ($48.2 + 3.2 > 50$).

5. EXPERIMENTAL EVALUATION

For the evaluation of the TTS scheduling strategy and the

Table 2: Computation of $barriers$ for sample schedule

	Conservative WCRT	Model-checking WCRT
$barrier(f_1, 1)$	$\{27.2, 8.45\}$	$\{26.6, 8.45\}$
$barrier(f_1, 2)$	$\{48.2, 3.2\}$	$\{46.7, 3.2\}$

design optimization method MCMSO, we performed simulations using synthetic task sets and the FMS application. We considered platforms with shared memory to which access is arbitrated based on a FCFS or RR policy. MCMSO was implemented in Wolfram Mathematica 9.0. The experimental setup for all simulations and the task generation algorithms are presented in detail in Sec. A.3.

Impact of barriers and resource contention. First, we evaluate the effects on schedulability when: (i) dynamic barriers in a time partitioning scheme and (ii) task interference on a shared resource are considered. For this purpose, we attempt to map/schedule randomly generated task sets under 3 alternative setups: (i) when inter-core interference is considered and the TTS sub-frames are dynamically initialized (MCMSO), (ii) when the TTS sub-frames are statically dimensioned based on the execution profiles of the tasks at their own CL (including inter-core interference), similarly to previous works for partitioned architectures, e. g., [29] (MCMSO, fixed sub-frames), and (iii) when inter-core interference is ignored and sub-frames are dynamic (MCMSO, no interference).

We use 500 synthetic task sets with 10 to 20 tasks and 2 CLs. For each generated task τ_i, computations and accesses under no resource contention take equal (randomly selected) time. The execution profiles of τ_i vary at different CLs, i. e., the access and execution time ranges of the phases at CL 2 are wider by a factor $Z_i \in [1, 4]$ than the respective ones at CL 1. We assume that all tasks with CL 1 are ignored at level of assurance 2 (degraded profile of zeros).

Figure 3 shows the fraction of task sets for which a schedulable implementation was found under the 3 setups, on architectures with 1, 2, 4, or 8 cores. The time budget of the optimizer was 25 minutes, but in most cases MCMSO converged to a solution within 10 minutes. As expected, the dynamism introduced by the use of synchronisation barriers and the fact that the low-criticality tasks are occasionally allowed to run in degraded mode (here, not run) in TTS lead to increased schedulability compared to the case with statically dimensioned sub-frames. This increase can be as high as 15% (4 cores). On the other hand, the results of Figure 3 show clearly the impact of resource contention on TTS schedulability. On multicore systems, schedulability could be increased up to 43% (8 cores) if resource sharing was eliminated, e. g., if the interfering tasks of $\mathcal{I}$ were mapped on the same core or such that they could not access the same memory bank at the same time. This can be critical information at design time, since by selecting a partitioned memory and mapping appropriately memory blocks to memory banks, the TTS schedulability can be significantly improved. It is also a strong argument for the need to consider the timing effects of resource contention in future multicore MC scheduling strategies. If one neglected these effects, up to 43% of the generated task sets would have erroneously been deemed schedulable.

Impact of using more cores under resource sharing. Next, we evaluate the gain in TTS schedulability as the number of cores in a multicore platform increases, while considering the effects of resource sharing. For this, we assume

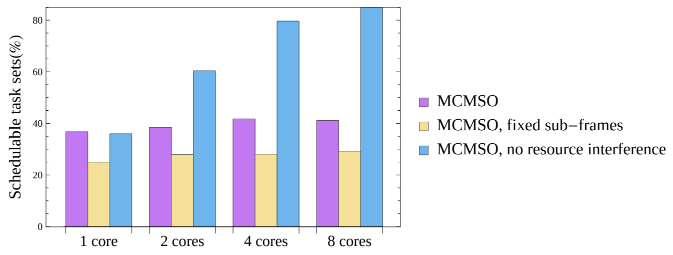


Figure 3: Schedulability under the 3 MCMSO setups for different number of cores

that in the 500 random task sets, the ratio of the minimum time required for memory accesses over the complete execution time of every task can be equal to: 0 (no accesses), 0.25, 0.5 (equal time for computations, accesses), 0.75 or 1 (no computation), i.e., we trade computation for accesses.

Figure 4 shows the fraction of schedulable task sets for the different scenarios, denoted as accessing time ratios (ATR), as the number of cores increases from 1 to 2, 4, and 8. It is interesting that the more memory-intensive the task sets are, the less gain we obtain by increasing the cores. Note that when ATR = 0.75 or 1, schedulability remains unchanged upon transition from single-core to multicore systems. Also in less memory-intensive cases (ATR = 0.5), schedulability may increase only up to 4 cores, but remains unchanged upon transition to 8 cores. This is probably opposed to what one would expect, given that more tasks (of the same CL) are able to execute in parallel. This is explained by the presence of shared resources because more concurrently executed tasks means also higher inter-core interference and thus, larger response times. We can conclude that in multicores with shared memories, the gain achieved by the increase of cores is limited unless there is a simultaneous increase in the memory bandwidth (reduction of access latency) or a reduction of the memory contention, e. g., by increasing the number of shared memory banks on the platform.

Comparison to existing MC scheduling strategies. Next, we evaluate the limitations posed by the (flexible) time-triggered implementation of TTS and their impact on schedulability. Hence, we compare TTS to more dynamic, state-of-the-art MC scheduling strategies, particularly the EDF-VD algorithm for single core [5] and its variant GLOBAL for multicores [20]. Since these algorithms do not consider resource sharing, comparison is based upon synthetic task sets that require no accesses. For task set generation we use the algorithm of [20] for 2 CLs. Per-task utilization U_i is selected uniformly from $[U_L, U_H] = [0.05, 0.75]$ and the ratio Z_i of the level-2 utilization to level-1 utilization is selected uniformly from $[Z_L, Z_H]$ =[1,8]. The probability that a task τ_i has $\chi_i = 2$ is set to $P = 0.3$. Period W_i is randomly selected from the set $\{100, 200, 300, 400, 500\}$. Because TTS cannot handle dynamic preemption, if the as-

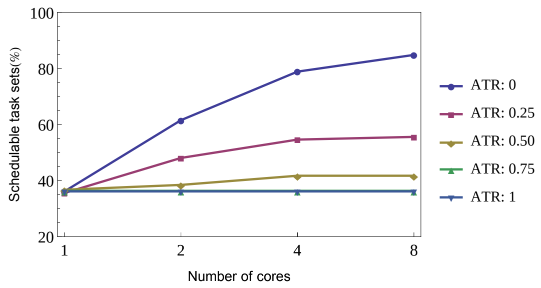
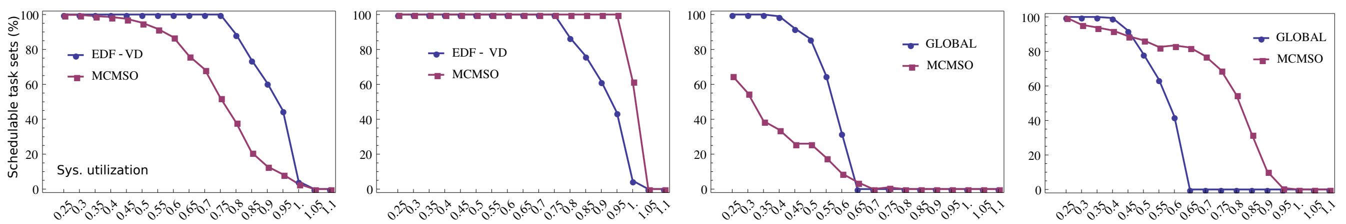


Figure 4: Schedulability vs. number of cores and accessing time ratio (ATR)



(a) Variable periods, fixed preemption, 1000 task sets (b) Equal periods, no preemption, 1000 task sets (c) Variable periods, fixed preemption, 100 task sets (d) Equal periods, no preemption, 1000 task sets

Figure 5: Schedulable task sets (%) vs. normalized utilization: MCMSO, EDF-VD ($m = 1$), GLOBAL ($m = 4$)

signed execution time of a task is larger than the maximum frame length of the TTS scheduling cycle, the task is split into sub-tasks, each "fitting" within a TTS frame.

Figures 5(a)-5(d) show the fraction of task sets that are deemed schedulable by the considered algorithms as a function of the ratio U_{sys}/m (normalized system utilization). U_{sys} is defined in [20] as follows:

$$U_{sys} = max\left(U^{\text{LO}}_{\text{LO}}(\tau) + U^{\text{LO}}_{\text{HI}}(\tau), U^{\text{HI}}_{\text{HI}}(\tau)\right), \qquad (5)$$

where $U^{\text{y}}_{\text{x}}(\tau)$ represents the total utilization of the tasks with CL x for their $y-$level execution profiles (LO≡1, HI≡2). To check schedulability of each randomly generated task set, we (i) apply MCMSO for TTS and (ii) check the sufficient conditions from [5, 20] for EDF-VD and GLOBAL.

In single-core systems, TTS faces two limitations compared to EDF-VD, i. e., the fixed preemption points and the time-triggered frames. EDF-VD is more flexible with scheduling task jobs as they arrive and can preempt them any time. The results of Figure 5(a) show that as the utilization increases, EDF-VD can schedule 0 up to 52.9% ($U = 0.85$) more MC task sets than TTS (avg: 17.9%). The impact of the TTS limitations on schedulability becomes even clearer if we repeat the experiment such that these limitations are avoided. This happens when all tasks have the same period ($W_i = 100$), hence the TTS cycle consists only of 1 frame. The corresponding results in Figure 5(b) exhibit now reverse trends, with TTS being able to schedule up to 57.2% ($U = 1.0$) more task sets than EDF-VD (avg: 10.5%). In fact, if we consider safety-critical applications with harmonic task periods, e. g., the FMS, the performance of TTS is comparable to that of EDF-VD (Sec. A.4), which is very important given that TTS was designed targeting at timing isolation rather than efficiency.

In multicores, GLOBAL performs more efficiently than TTS not only because of the previously discussed advantages, but also because it enables task migration. Namely, several jobs of the same task can be scheduled on different cores and a preempted job can be resumed on a different core. The results of Figure 5(c) show that the effectiveness of GLOBAL in finding admissible schedules for the generated task sets is up to 65% higher ($U = 0.40$) than for TTS. Recall, however, that the increased efficiency comes at the cost of ignoring the timing effects of shared resources which are not negligible especially in the presence of task migrations. If the limitations of TTS are avoided as before, the results (Figure 5(b)) are again reversed. Then, MCMSO finds a TTS schedule for up to 82.3% ($U = 0.65$) more task sets than GLOBAL. Sec. A.4 shows that schedulability is comparable when the task periods are harmonic. It follows that TTS, despite its imposed limitations for achieving timing isolation, can actually compete with state-of-the-art scheduling algorithms, which were designed with efficiency in mind.

Industrial system - FMS. Finally, to show the applicability of TTS scheduling to real-world industrial systems, we model the Flight Management System (FMS) application. FMS consists of 26 independent periodic tasks, characterized by 2 CLs. The tasks fit very well with the superblock model, hence we model each one as a sequence of an access (read), a computation and another access (write) phase. The phase parameters are selected randomly from pre-defined ranges, which we estimated depending on the functionalities of the corresponding tasks (Sec. A.3). For each task τ_i with $\chi_i = 2$, the execution time (access) ranges at CL 2 are wider by a factor of $Z_{ex} \in [1, 2]$ ($Z_{acc} \in [1, 4]$) than the ones at CL 1. Similarly, for each task τ_i with $\chi_i = 1$, we define a degraded execution profile such that the execution time (access) ranges in this profile are tighter by a factor of $D_{ex} \in [1, 2]$ ($D_{acc} \in [1, 10]$) than the ones at CL 1. The task periods of the FMS are known to be harmonic ($\{200, 1000, 5000\}$ ms), hence we dimension the TTS cycle with period $H = 5000$ ms and frames of equal length, 200 ms. The FMS features a computationally intensive task (flightplan computation, τ_{26}) with a worst-case execution time up to 800 ms at CL 2 (without considering accesses). We assume that we can split this task into 4, 5, 8 or 10 sub-tasks, each having the same access requirements as the original task and a fraction of its computation time.

We perform design space exploration with MCMSO to determine: (i) the minimum number of cores (1 to 8) for the FMS implementation, and (ii) the number of preemption points for τ_{26}. We evaluate schedulability of the FMS for the 4 alternative implementations of this task, based on $||barriers||_3$ for the corresponding best solution found by MCMSO. The lower the norm of a solution is, the higher the FMS schedulability. Note that every time MCMSO finds a schedulable implementation of FMS on m cores, exploration for the problem with $m + 1$ cores starts from that solution. In most cases, the optimizer converged to a solution in less than 25 minutes.

Figure 6 shows how the schedulability metric changes for one particular FMS instance as the number of cores increases, for different number of preemption points. The points within the dashed rectangle correspond to schedulable implementations. We observe that schedulability of an FMS implementation increases or remains stable as the number of cores increases. Also, more preemption points do not necessarily mean higher schedulability. This is because, as the number of preemptions increases, the required memory accesses can also increase due to the performed context switches. This is reflected by the higher schedulability of the implementation with 8 sub-tasks w.r.t. that with 10 sub-tasks. The results of Figure 6 are particularly useful for the FMS designer, as they show that a platform with at least 3 cores must be used and that the flightplan task must have at least 5 preemption points whereas having more than

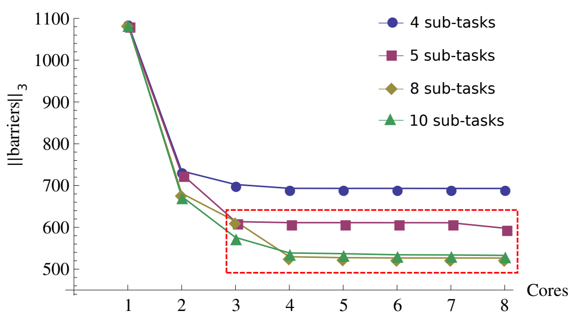


Figure 6: Flight Management System - DSE

8 is not beneficial.

6. CONCLUSION & FUTURE WORK

A global time-triggered scheduling approach with barrier synchronization (TTS) is proposed. It considers periodic mixed-criticality task sets executed on resource-sharing multicores. TTS enables only tasks of the same criticality level to be executed concurrently in order to guarantee their timing properties at a particular level of assurance, a necessary property for the certification of MC systems. TTS tries to bridge the space between strict partitioning mechanisms for timing isolation (industrial practice for safety-critical applications) and efficient MC scheduling algorithms. Moreover, TTS can take advantage of state-of-the-art interference analysis methods for multicore resource-sharing systems. As the experimental results show, TTS can schedule task sets without over-provisioning resources (increased schedulability compared to static partitioning approaches). At the same time, its efficiency is not severely compromised when compared to more dynamic MC scheduling strategies. Applicability has been validated with an industrial avionics application. This confirms that TTS is a potential solution to the problem of MC scheduling on multicores, where resource (e. g., memory) sharing among several cores cannot be eliminated. TTS is currently being implemented in an industrial setup for evaluating its runtime overhead.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 288175 (CERTAINTY project).

7. REFERENCES

[1] RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.

[2] R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In *Intl. Colloquium on Automata, Languages and Programming*, pages 322–335, 1990.

[3] 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.

[4] ARINC. ARINC 653-1 avionics application software standard interface. Technical report, 2003.

[5] 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*, pages 145–154, 2012.

[6] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *Mathematical Foundations of Computer Science*, pages 90–101, 2010.

[7] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *RTSS*, pages 3–12, 2011.

[8] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, pages 13–22, 2010.

[9] D. Dasari, B. Andersson, V. Nelis, S. Petters, A. Easwaran, and J. Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *TrustCom*, pages 1068 –1075, 2011.

[10] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305–331, 2010.

[11] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *ECRTS*, pages 135–144, 2012.

[12] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, and P. Gai. Time and memory tradeoffs in the implementation of AUTOSAR components. In *DATE*, pages 864 –869, 2009.

[13] 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*, pages 63–72, 2012.

[14] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *RTSS*, pages 13–23, 2011.

[15] O. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *TrustCom*, pages 1051–1059, 2011.

[16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[17] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, 14(3):193–227, 2010.

[18] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* 2001.

[19] H. Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *RTSS*, pages 183–192, 2010.

[20] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *ECRTS*, pages 166–175, 2012.

[21] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, pages 339–349, 2010.

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

[23] R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *ECRTS*, pages 309–320, 2012.

[24] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *RTSS*, pages 221–231, 2008.

[25] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, pages 741–746, 2010.

[26] S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, pages 759–764, 2010.

[27] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *RTAS*, pages 215–224, 2010.

[28] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, pages 213–222, 2011.

[29] D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *RTSS*, pages 24–33, 2011.

[30] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, pages 239–243, 2007.

[31] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, 2009.

[32] 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*, pages 299–308, 2012.

A. SUPPLEMENT

A.1 Motivational Example (FMS)

The flight management system (FMS) from the avionics domain is responsible for functionalities, such as the localization of an aircraft based on periodically acquired sensor data, the computation of the flightplan that guides the autopilot, the detection of the nearest airport, etc.

Tasks in this system are independent and communicate through double buffering. Some of them regularly read a navigation database. Due to the database size (magnitude of hundreds of MB), it is unrealistic to assume that each core can retain a copy of all required data in its local memory for most COTS multicores. So, the inter-core interference on the global memory path cannot be neglected. The superblock model of execution fits very well with the FMS tasks, since most of them fetch data from the memory (sensor readings, database) upon their triggering, process them and write back the results (localization information, computed flightplan) before their completion.

Besides periodic tasks in the FMS, there are asynchronous and restartable tasks. Asynchronous tasks are initiated by the pilot, but their invocation is limited by a known maximum frequency. Therefore, they can be modeled as periodic. Restartable tasks are computationally intensive tasks, related to the generation of the flightplans. These tasks need to be stopped and restarted if they overrun. They are also triggered at a known maximum frequency, so they are modeled as periodic for scheduling analysis. Special attention is needed for their scheduling so that they do not exhaust the computational resources.

Depending on the criticality of the corresponding functionalities, the FMS tasks are classified into 2 design assurance levels, DAL-B and DAL-C from the DO-178B standard for certification of airborne systems [1]. Note that for the FMS, it is not acceptable to drop lower-criticality tasks in order to guarantee the schedulability of higher-criticality tasks, as usually assumed in the MC scheduling literature. In practice, most lower-criticality tasks must be always executed, at least in (a pre-defined) degraded mode, for the safe operation of the aircraft.

Assume that the FMS tasks need to be mapped and scheduled on a multicore platform with private memories for each core and a shared memory, where the database and the shared buffers are maintained. If the accesses to the shared memory and the resulting timing interference were ignored, a solution would be to map tasks of the same CL on the same core and apply conventional real-time scheduling algorithms. Or, for more efficient workload distribution, map tasks of mixed CLs on any core and apply an existing MC-scheduling algorithm, e. g., [20,23]. In the presence of shared memory, these solutions do not enable composable certifiability though, since tasks of different CLs can interfere with each other upon accessing the memory. TTS, on the other hand, provides the required timing isolation among different CLs.

A.2 MCMSO heuristics

A variation of the basic SA heuristic algorithm, which was implemented for the MCMSO prototype is listed as pseudocode under Algorithm 1. The algorithm receives as inputs the initial temperature T_0, the temperature decreasing factor $a \in (0, 1)$, $Fail_{max}$ which defines the maximum number of consecutive variations with no cost improvement that can be checked for a particular temperature, T_{final} which is a stopping criterion in terms of the final temperature, and t_{max} which is a stopping criterion in terms of search time (computational budget). Its execution returns the best solution that was found for the selected temperature parameters, in the given time.

Algorithm 1 MCMSO-SA

```
Input: $T_0$, $a$, $Fail_{max}$, $T_{final}$, $t_{max}$
Output: $S_{best}$, $Cost_{min}$

 1: $S \leftarrow$ GenerateInitialSolution()
 2: $S_{best} \leftarrow S$
 3: $Cost_{min} \leftarrow$ Cost($S$)
 4: $T \leftarrow T_0$
 5: $FailCount \leftarrow 0$
 6: t $\leftarrow$ StartTimer()
 7: while $t < t_{max}$ and $T > T_{final}$ do
 8:     $S' \leftarrow$ RandomVariate($S$)
 9:     if $e^{-\left(\mathbf{Cost}(S')-\mathbf{Cost}(S)\right)/T} \geq$ Random(0,1) then
10:         $S \leftarrow S'$
11:     end if
12:     if Cost($S'$) $< Cost_{min}$ then
13:         $S_{best} \leftarrow S'$
14:         $Cost_{min} \leftarrow$ Cost($S'$)
15:         $FailCount \leftarrow 0$
16:     else
17:         $FailCount \leftarrow FailCount + 1$
18:     end if
19:     if $FailCount == Fail_{max}$ then
20:         $T \leftarrow a \cdot T$
21:         $S \leftarrow S_{best}$
22:         $FailCount \leftarrow 0$
23:     end if
24: end while
```

Functions *GenerateInitialSolution* and *RandomVariate* are specified such that only meaningful solutions (fulfilling the criteria of Sec. 4.2) are considered. Therefore, before applying the algorithm, one needs to compute for each task job, the earliest and latest TTS frame, in which the job can be scheduled. Function *RandomVariate* generates a new solution S' by selecting randomly a task of τ and performing non-deterministically a core, frame or alternative variation. The probability of choosing each of the three variations is defined in *RandomVariate*. The new solution is accepted, independently of whether its cost is higher or lower than that of S on condition that $e^{-\left(Cost(S')-Cost(S)\right)/T}$ is no lesser than a randomly selected real value in (0,1). Additionally, the cost of S' is compared to the currently best observed cost, $Cost_{min}$. If the former is lower than $Cost_{min}$, the new solution and its cost are stored, even if the transition to S' was not admitted.

As far an the annealing schedule is concerned, the temperature T is reduced geometrically, with factor a, every time a sequence of $Fail_{max}$ new solutions are checked, none of which has a lower cost than the best observed, $Cost_{min}$. In this case, the temperature is reduced to enable a finer exploration of the search space close to the currently best solution S_{best}. This solution is assigned to the current state S and the exploration continues similarly, until the lowest temperature T_{min} is reached or the computational budget t_{max} is exhausted.

For the simulations, described in Sec. 5 and Sec. A.3, MCMSO-SA was called with the following input parameters: $a = 0.8$, $Fail_{max} = 100$, $T_{final} = 0.1$. The initial temperature T_0 was selected in every case, by performing few random transitions and determining the average objective function change. The time budget varied as described for each experiment. Finally, the probabilities of *Random-Variate* selecting a core or a frame variation were set to 0.15 and 0.85, respectively (no alternative variation needed).

A.3 Experimental Setup

This section presents in detail the simulation setup for the case studies presented in Sec. 5. All simulations were run on computers with Intel Xeon CPUs at 2.9 GHz or AMD Opteron CPUs at 2.6 GHz and 8 GB of RAM.

Impact of barriers and resource contention. The experiment was conducted upon randomly generated task sets, characterized by 2 CLs. The input parameters for the generation of one task set are:

- n: Number of tasks.
- $[U_L, U_U]$: Lower and upper bound on per-task utilization. The utilization is uniformly selected from this range ($U_L > 0, U_U \leq 1$) and is the same for all tasks of a task set. It is defined as the fraction of worst-case execution time of a task over its period (Eq. 6), when execution time is considered at the task's own CL and the task needs **no** resource accesses:

$$u_i = \frac{\sum_{s_{i,j} \in S_i} ex_{i,j}^{max}(\chi_i)}{W_i} \qquad (6)$$

  where S_i is the set of superblock phases of task τ_i. For example, if a task τ_i has period $W_i = 100$ ms and the per-task utilization of the corresponding task set is $U = 0.2$, then the sum of worst-case computation time over τ_i's execution phases will be $0.2 \cdot 100 = 20$ ms at CL χ_i.
- $[Z_L, Z_U]$: Lower and upper bound on degree of pessimism of the tools used for the certification of high-criticality tasks. The degree of pessimism is defined as the ratio of a task's utilization at CL 2 (high) to its utilization at CL 1 (low). It is selected uniformly from the above range and independently for each task of a task set. For example, if a task τ_i with $\chi_i = 2$ has utilization $U = 0.2$, its utilization at CL 1 will be $\frac{U}{Z_i}$ with $Z_i \in [Z_L, Z_H]$.
- W_{set}: The set of task periods. A period is randomly selected from this set for each task.
- P_{crit}: The probability that a task τ_i is characterized by the highest CL ($\chi_i = 2$).
- T_{acc}: The latency of a memory access.

For each task τ_i of a generated task set, we assume the following:

- The task is defined by one accessing and one execution phase.
- If the task is of lower criticality ($\chi_i = 1$), its degraded mode is equivalent to no execution. Namely, when certifying the task set at level of assurance 2, the low-criticality tasks can be ignored.
- Several versions of the task can be generated depending on the *accessing time ratio* (ATR). This parameter defines the fraction of time the task spends on resource accessing over its total execution (sum of accessing and computation) time, when no resource contention exists. Depending on the ATR value, generation of a task's phase parameters is described in pseudocode form in Listing 2. The execution profile of τ_i C_i at level ℓ is defined by two superblock phases, denoted by the tuples $\{\mu^{min}, \mu^{max}, 0, 0\}$ (accessing phase) and $\{0, 0, ex^{min}, ex^{max}\}$ (execution phase), respectively.

Algorithm 2 TaskGenerator

```
Input: $\chi_i$, $W_i$, $U$, $Z_i$, ATR, $T_{acc}$
Output: $C_i(1)$, $C_i(2)$

1:  if $\chi_i = 1$ then
2:      $\text{exec\_time\_tot}(1) \leftarrow u \cdot W_i$
3:      $\mu(1) \leftarrow \left\lceil \frac{\text{ATR} \cdot \text{exec\_time\_tot}(1)}{T_{acc}} \right\rceil$
4:      $ex(1) \leftarrow \text{exec\_time\_tot}(1) - \mu(1) \cdot T_{acc}$

5:      $C_i(1) \leftarrow \{\{\mu(1), \mu(1), 0, 0\}, \{0, 0, ex(1), ex(1)\}\}$
6:      $C_i(2) \leftarrow \{\{0, 0, 0, 0\}, \{0, 0, 0, 0\}\}$
7:  else
8:      $\text{exec\_time\_tot}(2) \leftarrow u \cdot W_i$
9:      $\mu(2) \leftarrow \left\lceil \frac{\text{ATR} \cdot \text{exec\_time\_tot}(2)}{T_{acc}} \right\rceil$
10:     $ex(2) \leftarrow \text{exec\_time\_tot}(2) - \mu(2) \cdot T_{acc}$

11:     $\text{exec\_time\_tot}(1) \leftarrow \frac{u \cdot W_i}{z_i}$
12:     $\mu(1) \leftarrow \left\lceil \frac{\mu(2)}{z_i} \right\rceil$
13:     $ex(1) \leftarrow \text{exec\_time\_tot}(1) - \mu(1) \cdot T_{acc}$

14:     $C_i(1) \leftarrow \{\{\mu(1), \mu(1), 0, 0\}, \{0, 0, ex(1), ex(1)\}\}$
15:     $C_i(2) \leftarrow \{\{\mu(1), \mu(2), 0, 0\}, \{0, 0, ex(1), ex(2)\}\}$
16: end if
```

The parameter values for the simulations that yielded the results of Fig. 3 and 4 were set as follows: $n = 10$ (300 task sets), $n = 15$ (150 task sets) or $n = 20$ (150 task sets), $U_L = 0.02$, $U_U = 0.2$, $Z_L = 1$, $Z_U = 4$, $W_{set} = \{100, 200, 400, 500\}$, $P_{crit} = 0.5$, $T_{acc} = 0.5$. For each task set, we considered 5 different versions of its tasks for the different values of parameter ATR $\in \{0, 0.25, 0.5, 0.75, 1\}$. The TTS scheduling cycle was dimensioned so that the frames would be equally sized with length given by the greater common divisor of the tasks' periods. Note that based on the period set W_{set}, the maximum possible period of the TTS cycle (hyper-period of tasks) is $H = 2000$ ms and the maximum number of TTS frames is 20.

For each task set, the MCMSO optimizer was called in total 40 times; 2 times for each of the 5 task configurations (ATR parameter) and each of the 4 platform configurations (with 1, 2, 4 or 8 cores). The first time it was called, the optimizer took the timing effects of resource contention into account during exploration (MCMSO), while the second time it ignored them (MCMSO, no resource interference). To decide whether a found solution would be admissible had the sub-frames had fixed lengths (MCMSO, fixed sub-frames), we checked the condition of Eq. 7 for each solution found by MCMSO (1st call):

$$\sum_{i=1}^{L} barriers(f, i)_{L-i+1} \leq \mathcal{L}_f, \forall f \in \mathcal{F}. \qquad (7)$$

That is, a schedule is admissible with fixed sub-frames if for each frame f, the length of f is no lesser than the sum of its sub-frame lengths, when the latter are derived at their corresponding CL. For example, for the first sub-frame ($\ell = L$), we consider its length at level of assurance L, for the second sub-frame ($\ell = L-1$) at level $L-1$, etc. The index $(L-i+1)$ in Eq. 7 indicates the corresponding sub-frame (CL: i) within vector $barriers(f, \ell)$. Note that for condition 7, we do not consider the degraded mode of the tasks of lower CLs. That is because the sub-frames are dimensioned statically. If the degraded profile of some tasks was used during dimensioning, the corresponding tasks would be permanently executed in degraded mode, which is generally not acceptable in safety-critical systems, such as the FMS. Recall that under TTS scheduling, execution in degraded mode can occur rarely (depending on accuracy of the execution profiles at each level of assurance) and only for the duration of a TTS frame each time.

In all cases, the MCMSO optimizer uses the same cost function (see Sec. 4.2) during DSE. The time budget for each call was limited to 25 minutes, since it was observed that almost always the MCMSO converged to a solution within at maximum 10 minutes. Thus, we assumed that if no solution can be found in 25 minutes, no admissible solution exists at all.

Impact of using more cores under resource sharing. Same as above. The experiment was performed on the same randomly generated task sets.

Comparison to existing MC scheduling strategies. The goal of this experiment is to replicate the results of [20] in order to compare the ability of MCMSO to find schedulable implementations of MC task sets to that of EDF-VD on single cores or GLOBAL on multicores. Therefore, for the random task set generation, algorithm TaskGen (Fig. 4, [20]) was implemented and applied with parameters $U_L = 0.05$, $U_U = 0.75$, $Z_L = 1$, $Z_U = 8, P = 0.3$ (same as in Fig. 5, 6 of [20]). Moreover, we assumed that the (non-harmonic) periods of the generated tasks could be selected from the set $W_{set} = \{100, 200, 300, 400, 500\}$. Similarly to the previous experiment, the TTS cycle was dimensioned such that all frames have equal lengths, given by the greatest common divisor of the tasks' periods. Note that since the maximum per-task utilization is $U_i = 0.75$ and the maximum period $W_i = 500$ ms, it follows that a task can have a worst-case execution time of $0.75 \cdot 500 = 375$ ms at the highest CL. At the same time, the minimum possible frame length is 100 ms, implying that a task set containing the above task would be deemed not schedulable according to TTS. For those cases, we consider the existence of fixed preemption points. That is, if a generated task has a worst-case execution time greater than the TTS frame length, the task is split into so many sub-tasks such that each of them "fits" within a TTS frame. For instance, the previously mentioned task with execution time 375 ms would be split into 4 sub-tasks, each with execution time 93.75 ms if the TTS frame length was equal to 100 ms.

To evaluate the schedulability of a task set τ on single cores under EDF-VD, the (improved) sufficient condition from [5] is checked:

$$U^{\mathrm{HI}}_{\mathrm{HI}}(\tau) + U^{\mathrm{LO}}_{\mathrm{LO}}(\tau) \cdot \frac{U^{\mathrm{LO}}_{\mathrm{HI}}(\tau)}{1 - U^{\mathrm{LO}}_{\mathrm{LO}}(\tau)} \leq 1. \tag{8}$$

Similarly, for schedulability on multicores (m cores) under GLOBAL, the sufficient condition 9 from [20] must hold:

$$U^{\mathrm{LO}}_{\mathrm{LO}}(\tau) + \min\left(U^{\mathrm{HI}}_{\mathrm{HI}}(\tau), \frac{U^{\mathrm{LO}}_{\mathrm{HI}}(\tau)}{1 - 2 \cdot U^{\mathrm{HI}}_{\mathrm{HI}}(\tau)/(m+1)}\right) \leq \frac{m+1}{2}, \tag{9}$$

where $U^{\mathrm{LO}}_{\mathrm{LO}}$, $U^{\mathrm{LO}}_{\mathrm{HI}}$ and $U^{\mathrm{HI}}_{\mathrm{HI}}$ are defined as in Eq. 1 of [20]. Respectively, for schedulability under TTS, the condition of Eq. 3 must be validated. For this experiment, the MCMSO optimizer was given a time budget of 20 minutes, since it was observed that this was already a multiple of the amount of time needed until the simulated annealing heuristics converged to a solution.

The (normalized) system utilization U_{bound} (U_{bound}/m), against which schedulability is examined (x-axis in Fig. 4(a), 4(b), 4(c), 4(d)), is defined as in Eq. 8 of [20]. Note that this parameter increases from 0.25 to 1.10 in steps of 0.05. Each point in the figures was obtained by (i) randomly generating 1000 task sets, (ii) checking the schedulability of each one under the respective state-of-the-art algorithm and TTS, and (iii) calculating the fraction of schedulable task sets (y-axis) on single-core or quad-core platforms.

Industrial system - FMS. For this experiment, we looked into a subset of functionalities of the Flight Management System (Sec. A.1). For each considered task, we knew from an actual implementation of the FMS on a single-core system, its activation pattern (periodic - P, asynchronous - A or restartable - R), criticality level (DAL-B - 2 or DAL-C - 1), and period. This information is depicted in Table 3. Based on the implementation of each task and the data structures that it needs to read/update, we estimated lower and upper bounds on the number of required memory accesses upon triggering (data fetches) and before completion (write-backs) of the task, given that all data structures are maintained in the main memory. Moreover, we assumed that the computation time of a task cannot surpass 10% of its period (2% for periods $\geq$ 5 sec). Therefore, we set the lower and upper limits on access requests and computation times (in ms) that appear in Table 3. A range of values, e. g., of the form $\{\mu^{min}, \mu^{max}\}$ for resource accesses denotes these limits. Note that these ranges have not been derived by static analysis or profiling of the tasks' code, since the latter was not available. They are estimations based on the tasks' functionalities that were assumed for the purposes of the experiment. Static analysis of an FMS implementation on a specific multicore platform would probably yield different access request and computation time ranges.

As a next step, we generated randomly several instances of the FMS task set based on the following parameters:

- $[Z_{L,acc}, Z_{U,acc}]$: Lower and upper bound on degree of pessimism for the certification of high-criticality tasks, with regard to resource accesses. For instance, a task τ_i with $\chi_i = 2$ and an upper bound of $\mu^{max}(2)$ accesses will have an upper bound $\mu^{max}(1) = \lceil \mu^{max}(1)/Z \rceil$ accesses in its level-1 execution profile, where $Z \in [Z_{L,acc}, Z_{U,acc}]$. In our experiments, parameter Z was selected uniformly from the range $[1, 4]$ and was the same for all tasks of an FMS instance with CL 2.
- $[Z_{L,exec}, Z_{U,exec}]$: Same as before for the computation time of the tasks with CL 2. The degree of pessimism Z_i was selected uniformly from the range $[Z_{L,exec}, Z_{U,exec}] = [0, 1]$ and independently for each FMS task τ_i with $\chi_i = 2$.
- $[D_{L,acc}, D_{U,acc}]$: Lower and upper bound on fraction of memory accesses that are performed when tasks with CL

Table 3: FMS Parameters

Purpose	Task	Act.Pattern	CL	Period	Accesses (Fetch)	Exec.Time	Accesses (Write Back)
Sensor data acquisition	τ_1	P	2	200	{50,108}	{1,20}	{50,85}
	τ_2	A	2	200	-	{1,20}	{0,7}
	τ_3	A	2	200	-	{1,20}	{0,19}
	τ_4	A	2	200	-	{1,20}	{0,19}
	τ_5	A	2	200	-	{1,20}	{0,9}
Localization	τ_6	P	2	200	{50,127}	{1,20}	{10,18}
	τ_7	P	2	1000	{10,18}	{1,100}	{10,18}
	τ_8	P	2	5000	{20,35}	{1,100}	{0,1}
	τ_9	P	2	1000	{20,33}	{1,100}	{0,4}
	τ_{10}	A	2	200	-	{1,20}	{10,20}
	τ_{11}	A	2	1000	-	{1,100}	{0,3}
	τ_{12}	A	2	200	-	{1,20}	{0,3}
Flightplan management	τ_{13}	A	2	1000	{100,200}	{1,100}	{20,100}
	τ_{14}	A	1	1000	{100,200}	{1,100}	{20,100}
	τ_{15}	A	2	1000	{100,200}	{1,100}	{20,100}
	τ_{16}	A	1	1000	{100,200}	{1,100}	{20,100}
	τ_{17}	A	2	1000	{100,200}	{1,100}	{20,100}
	τ_{18}	A	2	1000	{100,200}	{1,100}	{20,100}
	τ_{19}	A	1	1000	{100,200}	{1,100}	{20,100}
	τ_{20}	A	1	1000	{100,200}	{1,100}	{20,100}
Flightplan computation	τ_{21}	P	2	1000	{0,3}	{1,100}	{0,3}
	τ_{21}	P	2	1000	{30,54}	{1,100}	{20,44}
	τ_{23}	R	2	5000	{200,300}	{700,800}	{100,180}
	τ_{23a}	R	2	5000	{200,300}	{175,200}	{10,45}
	τ_{23b}	R	2	5000	{200,300}	{140,160}	{10,36}
	τ_{23c}	R	2	5000	{200,300}	{87,100}	{10,23}
	τ_{23d}	R	2	5000	{200,300}	{70,80}	{10,18}
Guidance	τ_{24}	P	2	200	{0,10}	{1,20}	{0,1}
	τ_{25}	P	2	200	{0,10}	{1,20}	{0,1}
Nearest Airport	τ_{26}	P	1	1000	{100,134}	{1,100}	{200,322}

1 run in degraded mode. For instance,a task τ_i with $\chi_i = 1$ and an upper bound of $\mu^{max}(1)$ accesses will perform at maximum $\mu^{max}(2) = \mu^{max,deg} = \lceil \mu^{max}(1)/D \rceil$ accesses in its $C_{i,deg}$ execution profile, with $D \in [D_{L,acc}, D_{U,acc}]$. In our experiments, D was selected uniformly from the range $[1, 10]$, once for all tasks of an FMS instance with CL 1.

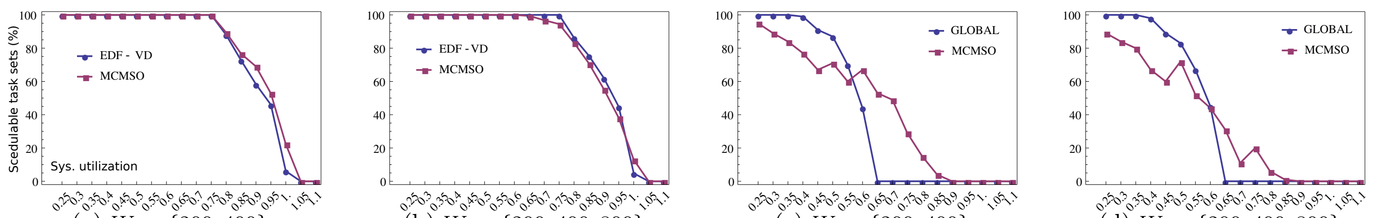
- $[D_{L,exec}, D_{U,exec}]$: Same as before for the computation time of the tasks with CL 1. Parameter D_i was selected uniformly from the range $[D_{L,exec}, D_{U,exec}] = [0, 1]$, independently for each FMS task τ_i with $\chi_i = 1$.

An FMS instance can be generated as follows. Let each task consist of 3 superblock phases; an accessing phase at the beginning, followed by a computation and another accessing phase. The minimum/maximum ranges of resource accesses and computation time for these phases at the task's own CL are generated randomly such that they are included in the given ranges of Table 3. The corresponding level-1 execution profile for tasks τ_i with CL 2 is derived based on the parameters Z (accesses) and Z_i (computation time). Respectively, the level-2 (degraded) execution profile for tasks τ_i with CL 1 is derived depending on parameters D (accesses) and D_i (computation time).

Following the previous procedure, we generated several FMS instances. For each instance, we considered 4 alternatives, for all potential implementations of the restartable task τ_{23} (4 dependent sub-tasks τ_{23a} or 5x τ_{23b} or 8x τ_{23c} or 10x τ_{23d}, see Table 3 for corresponding parameters). The MCMSO optimizer was called to find an implementation of each instance on systems with 1 to 8 cores and a shared memory with access latency $T_{acc} = 0.05$ ms. The optimizer was given a time budget of 35 minutes for the DSE and in practice, it was observed that it never needed more than 25 minutes to converge to a solution (admissible or not). Note that for this experiment, in order to boost the efficiency of the MCMSO optimizer, for systems with more than one core, the simulated annealing heuristics started not from a random task mapping, but from the solution found for the system with one less core. That is, if a solution S_3 was found for an FMS instance on a 3-core system, the DSE for a 4-core system started from solution S_3. If S_3 was admissible, then it would also be admissible on a 4-core system (where a core is not used). This way, the MCMSO started from a good initial solution ($c_1 > 0$), which it tried to improve.

A.4 Additional Experimental Results

Comparison to existing MC scheduling strategies. The following results complement the comparison to state-of-the-art MC scheduling strategies of Sec. 5. In particular, one more set of simulations is run for single-core ($m = 1$) and multicore ($m = 4$) systems, where the task sets have not equal nor random, but *harmonic* periods. In other words, every task period divides evenly every other (greater) period in W. This is rather common in the domain of safety-critical applications on which we are focusing, as witnessed e. g., in



(a) $W_i \in \{200, 400\}$ (b) $W_i \in \{200, 400, 800\}$ (c) $W_i \in \{200, 400\}$ (d) $W_i \in \{200, 400, 800\}$

	(a)		(b)		(c)		(d)
Min. Diff:	0%	Min. Diff:	-6.7%	Min. Diff:	-24%	Min. Diff:	-31%
Avg. Diff:	2.2%	Avg. Diff:	-1.2%	Avg. Diff:	3.8%	Avg. Diff:	-3.6%
Max. Diff:	16.2%	Max. Diff:	8.1%	Max. Diff:	53%	Max. Diff:	31%
Opt. Time:	26.8s	Opt. Time:	60.5s	Opt. Time:	89.7s	Opt. Time:	159.7s

Figure 7: Schedulable task sets (%) vs. normalized utilization: MCMSO, EDF-VD ($m = 1$), GLOBAL ($m = 4$), $U_L = 0.05, U_L = 0.75, Z_L = 1, Z_L = 8, P = 0.3$

the FMS application.

Similar to the simulation setup of Sec. 5, synthetic task sets with no resource accesses are generated according to the algorithm presented in [20] for 2 CLs. Per-task utilization U_i is selected uniformly from $[U_L, U_H] = [0.05, 0.75]$ and the ratio Z_i of the level-2 utilization to level-1 utilization is selected uniformly from $[Z_L, Z_H]$ =[1,8]. The probability that a task τ_i has $\chi_i = 2$ (high CL) is set to $P = 0.3$. Since TTS supports fixed preemption points, if the assigned execution time of a task is larger than the maximum frame length of the TTS scheduling cycle, we assume that the task is split into sub-tasks, each "fitting" within a TTS frame. The simulations have been executed for two harmonic period sets. In the first case, the task periods W_i are selected uniformly from $\{200, 400\}$ (2 periods) and in the second case, from $\{200, 400, 800\}$ (3 periods).

Figures 7(a)-7(d) show the fraction of task sets that are deemed schedulable by the algorithms under comparison (MCMSO and EDF-VD or GLOBAL) as a function of the normalized system utilization, U_{sys}/m. To check schedulability of each randomly generated task set, we (i) apply MCMSO for TTS and (ii) check the sufficient conditions of Eq. 8 and Eq. 9 for EDF-VD [5] ($m = 1$) and GLOBAL [20] ($m = 4$), respectively. The schedulable fraction has been computed for 1000 task sets per utilization point in Figures 7(a)-7(b) and for 100 task sets, accordingly, in Figures 7(c)-7(d). Below the figures, we show the minimum, average and maximum difference between the schedulable fraction under TTS and EDF-VD/GLOBAL ($Sched\_task\_sets(TTS) - Sched\_task\_sets(EDF - VD)$) across all utilization points, as well as the maximum time required by the MCMSO optimizer to converge to an optimal TTS solution.

The results show that in the case of harmonic task periods, schedulability under TSS and the selected state-of-the-art algorithms is comparable. In single-core systems, TTS can schedule up to 16.2% ($U = 1.0$) and on average 2.2% more task sets that EDF-VD when the periods are selected from $\{200, 400\}$. For the period set $\{200, 400, 800\}$, it can schedule up to 8.1% ($U = 1.0$) more task sets. EDF-VD performs on average better by 1.2% in this case.

In multicores, TTS seems to perform worse than GLOBAL for low system utilizations (up to $U = 0.55$), but it is more efficient in finding schedulable solutions as utilization increases. Specifically, for the period set $\{200, 400\}$, TTS can schedule up to 24% ($U = 0.45$) less task sets than GLOBAL and up to 53% ($U = 0.65$) more task sets. On average, it performs better than GLOBAL by 3.8%. Its performance is slightly degraded for the period set $\{200, 400, 800\}$. Then, TTS can schedule from 31% ($U = 0.4$) less to 31% ($U = 0.65$) more task sets than GLOBAL, but on average, schedulability under GLOBAL is higher by 3.6%.

Summarizing, the simulation results confirm that the performance of TTS is very close to that of EDF-VD for single-core systems. For multicores, TTS performs worse than GLOBAL for low system utilizations, but its relative capability in finding schedulable solutions increases with the utilization. On average, schedulability under TTS and GLOBAL is comparable. This is a very important conclusion since the assumption of harmonic periods is not very restrictive in the domain safety-critical real-time systems. Under this, TTS, despite being designed for effective timing isolation in MC environments, exhibits comparable performance to state-of-the-art scheduling methods which target mainly at efficiency.