Mapping Mixed-Criticality Applications on Multi-Core Architectures

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
Email: firstname.lastname@tik.ee.ethz.ch

***Abstract*—A common trend in real-time embedded systems is to integrate multiple applications on a single platform. Such systems are known as mixed-criticality (MC) systems when the applications are characterized by different criticality levels. Nowadays, multicore platforms are promoted due to cost and performance benefits. However, certification of multicore MC systems is challenging as concurrently executed applications of different criticalities 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 response times of applications. Recently, a MC scheduling strategy was proposed, which explicitly accounts for these effects. This paper discusses how to combine this policy with an optimization method for the partitioning of tasks to cores as well as the static mapping of memory blocks, i.e., task data and communication buffers, to the banks of a shared memory architecture. Optimization is performed at design time targeting at minimizing the worst-case response times of tasks and achieving efficient resource utilization. The proposed optimization method is evaluated using an industrial application.**

I. Introduction

As a result of the prevalence of multicore systems in the electronics market, the field of embedded systems experiences an unprecedented trend towards integrating multiple applications into a single platform. This applies even to real-time embedded systems for safety-critical domains, such as avionics and automotive. Applications in these domains, however, are usually characterized by several criticality levels (CLs), known as *Safety Integrity Levels* (SIL) or *Design Assurance Levels* (DAL), which express the required protection against failure.

For the integration of *mixed-criticality* (MC) applications into a common platform, the existing certification standards require complete timing *isolation* among applications of different criticalities. For this, system designers rely mainly on partitioning mechanisms, e.g., based on the ARINC-653 standard [1]. No existing standards, however, specify how isolation is achieved when several cores share platform resources, which is a common practice for efficiency and cost reasons. Obviously, if several cores access synchronously e. g., a memory bus, timing interference among applications of different criticalities cannot be avoided. Also, the resulting delays cannot be always bounded, since the certification authorities for applications of a particular CL typically do not possess any information about applications of lower CL that are co-hosted on the platform.

In [2], Giannopoulou et al. suggest a scheduling policy for resource-sharing multicores, which prevents timing interference among applications of different CLs. This is achieved by allowing only a statically known set of applications of the *same* criticality to be executed across the cores at any time. Scheduling is implemented through flexible time triggering on the core level with static and dynamic barriers on the global level. This way timing isolation is preserved despite resource sharing, without any need for hardware support.

In this paper, we extend the work of [2] with the focus of design optimization under the suggested scheduling policy. The main contributions can be summarized as follows:

- We introduce an architecture abstraction for memory-sharing systems, where cores contend for access at the level of memory banks. This abstraction describes faithfully modern commercial platforms, such as Kalray MPPA-256 [3].
- We propose a heuristic approach for finding a mapping of mixed-criticality task sets to multicores and a partitioned schedule, such that the real-time requirements of the tasks are met and the workload is balanced among the cores.
- We propose a simulated annealing-based algorithm for statically mapping the task data and the communication buffers to global memory banks, such that the effect of bank sharing on the task response times is minimized.
- Since the above two optimization problems are interdependent, we propose two possible approaches for integrated design optimization.
- We evaluate applicability and efficiency of the design optimization approaches using a real-world avionics application.

Related Work Scheduling of mixed-criticality applications is a research field attracting increasing attention nowadays. Vestal introduced the currently dominating MC task model in [4]. The first MC scheduling algorithms for multicores appeared recently, among others in [5]–[7]. To comply with the certification requirements for strict timing isolation, Anderson et al. proposed scheduling MC tasks on multicores, by adopting different strategies for different CLs and utilizing a bandwidth reservation server [8]. Tamas-Selicean and Pop presented an optimization method for task partitioning and time-triggered scheduling on multicores [9], complying with the ARINC-653 standard [1]. Most of these works, however, ignored the interference when tasks access synchronously shared platform resources and its effect on schedulability. The scheduling strategy in [2] seems to be one of the first strategies to explicitly consider this, while ensuring timing isolation on global level.

For bounding interference on the shared memory path in MC settings, Yun et al. proposed a software-based memory throttling mechanism. In [10], the cores, where low-criticality tasks are executed, are assigned a limited memory budget so that schedulability of the high-criticality tasks is guaranteed while the performance impact on the low-criticality tasks is kept minimal. Despite bounding the timing effect of interference, restricting mapping to cores based on the tasks' CL may lead to inefficient system utilization. Similarly, the authors of [11], [12] suggested novel memory controller designs for mixed hard real-time and soft real-time systems. These methods require special hardware support as opposed to [2], where task scheduling prevents MC interference on the shared memory path.

On the topic of memory bank partitioning, recent works [13], [14] proposed heuristics for mapping data of different application threads to DRAM banks in order to reduce the average thread execution times. Contrary to our work, these methods do not provide any real-time guarantees. Liu et al. implemented in [15] a bank partitioning mechanism in terms of modifying the OS memory management system to adopt a custom page-coloring algorithm for the data allocation to banks. A similar mechanism could be used to impose the memory mapping decisions of our design optimization method. Closer to our objective lies [16], where the authors rely on a DRAM controller to partition the data of real-time applications to banks, such that interference on bank level is eliminated. This controller could provide the required timing isolation for MC systems and enable a decrease in the tasks' response times. However, it is a hardware solution. Note, also, that the aforementioned results consider partitioning of the private task data and not of shared data, on which interference cannot be eliminated.

II. System Model

This section defines the task and architecture models. The task model is an extension of the established MC model [4], accounting also for memory access. For the architecture model, special emphasis is put on the shared memory subsystem.

Task Model. We consider mixed-criticality periodic task sets $\tau = \{\tau_1, \dots, \tau_n\}$ with criticality levels between 1 (lowest) and L (highest). A task is characterized by a 4-tuple $\tau_i = \{W_i, \chi_i, \mathbf{C}_i, C_{i,deg}\}$, 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 $C_i(\ell) = (e_i^{max}(\ell), \mu_i^{max}(\ell))$ represents an upper bound on the execution time and number of memory accesses of τ_i at level of assurance $\ell \leq \chi_i$,
- $C_{i,deg}$ is a special execution profile for the cases when τ_i ($\chi_i < L$) runs in *degraded mode*. This profile corresponds to the minimum required functionality of τ_i so that no catastrophic effect occurs in the system. If execution of τ_i can be aborted without catastrophic effects, $C_{i,deg} = (0, 0)$.

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., $D_i = W_i$. Also, the parameters of $C_i(\ell)$ are monotonically increasing for increasing ℓ. The respective bounds can be obtained by different tools. For instance, at the lowest level of assurance ($\ell = 1$), the system designer may extract them by profiling and measurement. At higher levels, certification authorities may use static analysis tools with more and more conservative assumptions as the required confidence increases. Note that the execution profile $C_i(\ell)$ for each task τ_i is derived only for $\ell \leq \chi_i$. For $\ell > \chi_i$, there is no valid execution profile since certification at level ℓ ignores all tasks with a lower CL. At runtime, if a task with CL greater than χ_i requires more resources than initially expected, τ_i may run in degraded mode. Then, $C_{i,deg}$ describes its execution profile.

For any MC periodic task set τ, we seek a scheduling strategy which can provide an admissible schedule at all levels of assurance. A schedule is *admissible* at level ℓ if and only if:

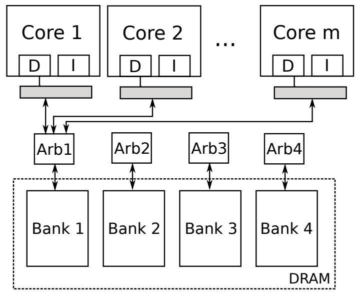

Figure 1. Memory Subsystem Architecture

- 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 w.r.t. 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 w.r.t. execution profile $C_{i,deg}$.

Multicore Resource-Sharing Architecture. We consider a set $\mathcal{P}$ of m processing cores, $\mathcal{P} = \{p_1, \dots, p_m\}$, Here, the cores are identical but our approach can be generalized to heterogeneous platforms. Each core in $\mathcal{P}$ has access to a private cache memory (we restrict our interest to data caches) and to a shared global memory. The shared (DRAM) memory is organized in several banks. Each bank must have a sequential address space (not interleaved among banks) to limit potential inter-task interference. Under this assumption, two concurrently executed tasks on different cores can perform parallel accesses to the shared DRAM without delaying each other provided that they access different banks. We assume that each memory bank has a dedicated request arbiter. Also, each core has a private path (bus) to the shared memory. The private paths of the cores are connected to all bank arbiters, as depicted abstractly (for Arb1) in Figure 1. This model of the memory subsystem seems to fit well commercial platforms, such as the Kalray MPPA-256 [3].

For the bank arbitration, we consider the policies of first-come first-served (FCFS) and round-robin (RR). However, other policies can also be modeled. We assume that only one core can access a bank at a time and that once granted, a bank access is completed within a fixed time interval, T_{acc} (same for read/write operations and for all banks). In the meanwhile, pending requests to the same bank from other cores stall execution on their cores until they are served. Note that we consider hardware platforms without timing anomalies, such as the fully timing compositional architecture [17], where execution and communication times can be decoupled.

We use a graph representation of the potential inter-task interferences due to memory contention, i.e., the *memory interference graph* (MIG) $\mathcal{I}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{V}_T \cup \mathcal{V}_{BL} \cup \mathcal{V}_B$. $\mathcal{V}_\tau$ represents all tasks in τ, $\mathcal{V}_{BL}$ all memory blocks BL accessed by τ, i.e., the task data and communication buffers, and $\mathcal{V}_B$ all DRAM banks B. Each memory block (bank) node is annotated with a corresponding size (capacity) in bytes. $\mathcal{I}$ is composed by two sub-graphs: (i) the bipartite graph $\mathcal{I}_1(\mathcal{V}_T \cup \mathcal{V}_{BL}, \mathcal{E}_1)$, where an edge $e \in \mathcal{E}_1$ from $\tau_i \in \mathcal{V}_T$ to $bl_j \in \mathcal{V}_{BL}$ with weight $w(e)$ implies that task τ_i performs at maximum $w(e)$ accesses to memory block bl_j per execution, and (ii) the bipartite graph $\mathcal{I}_2(\mathcal{V}_{BL} \cup \mathcal{V}_B, \mathcal{E}_2)$, where an edge $e \in \mathcal{E}_2$ from $bl_j \in \mathcal{V}_{BL}$ to $b_k \in \mathcal{V}_B$ denotes the allocation of memory block bl_j in exactly one memory bank b_k. Note that the weighted sum over all

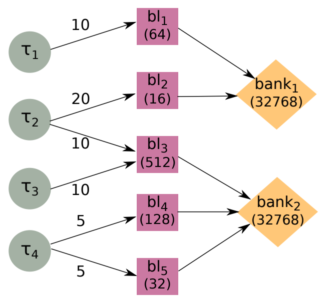

Figure 2. Memory Interference Graph

outgoing edges of a task τ_i equals the memory access bound of its execution profile at its own CL, i.e., $\mu_i^{max}(\chi_i)$. Figure 2 presents a possible MIG for a set of four tasks, accessing in total five memory blocks. The memory blocks can be allocated in two DRAM banks. Ellipsoid, rectangular and diamond nodes denote tasks, memory blocks, and banks, respectively.

Given a MIG $\mathcal{I}$, any two tasks τ_i and τ_j are defined as *interfering* if and only if $\exists k, l, r \in \mathbb{N}^+ : (\tau_i, bl_k) \in \mathcal{E}_1$, $(\tau_j, bl_l) \in \mathcal{E}_1$ and $(bl_k, b_r) \in \mathcal{E}_2$, $(bl_l, b_r) \in \mathcal{E}_2$, i.e., they access blocks in the same bank. In Figure 2 tasks τ_1 and τ_2 are interfering, whereas τ_1 and τ_3 or τ_4 are not. Interfering tasks can delay each other when executed in parallel.

Note that the binding of tasks to processing cores, $\mathcal{M}_\tau : \tau \rightarrow \mathcal{P}$ and the mapping of memory blocks to banks, $\mathcal{M}_{mem} : BL \rightarrow B$ ($\mathcal{E}_2$ of $\mathcal{I}$), are not given, but defined by our method. In particular, the problem that we are addressing can be formulated as follows. Given: (i) a periodic MC task set τ, (ii) an architecture consisting of cores $\mathcal{P}$ with a shared memory, and (iii) a memory interference graph $\mathcal{I}$ with undefined edge set $\mathcal{E}_2$; Determine: (i) the binding $\mathcal{M}_\tau$ of tasks to cores, (ii) the schedule $\mathcal{S}$ on the cores, and (iii) the mapping $\mathcal{M}_{mem}$ of memory blocks to banks, such that all tasks meet their MC real-time requirements at all levels of assurance, the workload is balanced among the cores, and the memory bank capacities are not violated.

III. Mixed-Criticality Scheduling

This section discusses briefly the Flexible Time-Triggered and Synchronisation-based (FTTS) MC scheduling policy. The reader is referred to [2] for a more detailed presentation.

The non-preemptive FTTS scheduling policy combines time and event-triggered task activation. A global FTTS schedule repeats over a *scheduling cycle* equal to the hyper-period H of the tasks in τ. The scheduling cycle consists of fixed-size *frames* (set $\mathcal{F}$). Each frame is divided further into L flexible-length *sub-frames*. The beginning of frames and sub-frames is synchronized among all cores. The frame lengths can differ, but they are bounded by the minimum period in τ. Each sub-frame (except the first of a frame) starts once all tasks of the previous sub-frame complete execution across all cores. Synchronisation is achieved dynamically via a barrier mechanism. Each sub-frame contains only tasks of the same CL. Note that the sub-frames within a frame are ordered in decreasing order of their CL and that within a sub-frame, tasks are scheduled sequentially on each core following a predefined order. An illustration of a FTTS schedule $\mathcal{S}$ is given in Figure 3 for seven tasks, a hyper-period of $H = 200$ ms, four frames of equal lengths (50 ms), each with $L = 2$ sub-frames. A cycle of $\mathcal{S}$ includes H/W_i invocations of each task τ_i, i.e., the number of jobs of τ_i that arrive within a hyper-period.

At runtime, the length of each sub-frame varies based on the different execution times and accessing patterns that the concurrently executed tasks exhibit. E.g., in Figure 3, the first sub-frame of f_1 finishes earlier when τ_1, τ_2 run w.r.t. their level-1 profiles (cycle 1) than when at least one task runs w.r.t. its level-2 profile (cycle 2). The sub-frame worst-case lengths can be computed offline for each schedule by applying worst-case response time (WCRT) analysis under memory contention. Function $barriers : \mathcal{F} \times \{1, \ldots, L\} \rightarrow \mathbb{R}^L$ defines the worst-case length of all sub-frames in a frame, for a particular level of assurance. We denote the worst-case length of the i-th sub-frame of f at level ℓ as $barriers(f, \ell)_i$. Note that ℓ corresponds to the highest level execution profile that the tasks of f exhibit at runtime.

Runtime behavior. Given an admissible FTTS schedule $\mathcal{S}$ and the *barriers* function, the scheduler manages task execution on each core within each frame $f \in \mathcal{F}$ as follows (init., $\ell_{max} = 1$):

- For the i-th sub-frame, the scheduler triggers sequentially the corresponding jobs. Upon completion of the jobs' execution, it signals the event and waits until the remaining cores reach the barrier.
- Let the elapsed time from the beginning of the i-th sub-frame until the barrier synchronisation be t. Given ℓ_{max}:
$$\ell_{max} = \max\{\underset{\ell \in \{1,\ldots,L\}}{\arg\min} \{t \leq barriers(f, \ell)_i\}, \ell_{max}\},$$
the scheduler will trigger jobs in all next sub-frames such that tasks with CL lower than ℓ_{max} run in degraded 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 affects only the current frame.

Admissibility. Let $\mathcal{S}$ be a FTTS schedule constructed such that all H/W_i jobs of each task $\tau_i \in \tau$ are scheduled on the same core within their release times and deadlines. $\mathcal{S}$ is ℓ-admissible if and only if it fulfils the following condition:

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

where $\mathcal{L}_f$ denotes the length of frame f. If the condition holds for all frames $f \in \mathcal{F}$, all scheduled jobs in S can meet their deadlines at level of assurance ℓ. If it holds for all levels $\ell \in \{1, \cdots, L\}$, it follows that schedule $\mathcal{S}$ is admissible.

IV. Mixed-Criticality Design Optimization

This section suggests approaches for task and memory mapping optimization for a MC task set scheduled under FTTS. For each optimization problem, we assume an existing solution to the other one. Finally, we show how to solve both optimization problems in an integrated manner.

A. Task Mapping ($\mathcal{M}_\tau + \mathcal{S}$) Optimization

A possible approach to scheduling and task mapping optimization was suggested in [2] and is summarized below.

This approach implements a heuristic method based on simulated annealing (SA) [18]. Initially, it selects a dimensioning of the FTTS cycle and frame lengths and generates a random

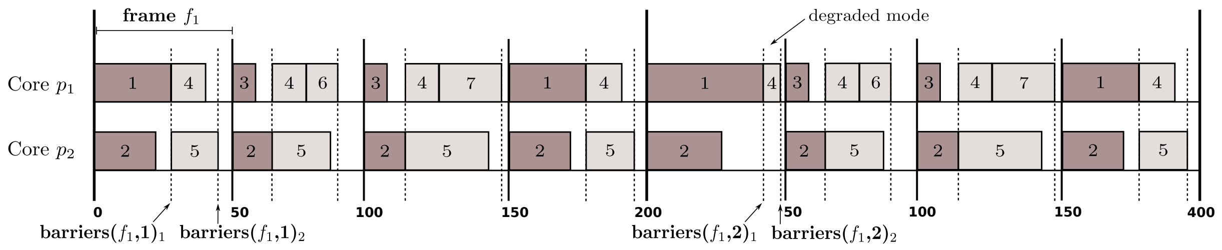

Figure 3. Global FTTS schedule for 2 cycles (dark annotation: CL 2, light: CL 1)

task mapping solution. In this solution, all jobs released by the tasks in τ within a hyper-period H must be scheduled, with every job scheduled in a frame between its release time and absolute deadline. By applying SA, the design space for task mapping is explored. Particularly, new solutions are found by applying two possible variations with certain probabilities: (i) re-mapping all jobs of a randomly selected task to a different core or (ii) re-allocating one job of a randomly selected task to a different FTTS sub-frame. Design space exploration (DSE) terminates when the algorithm converges to a solution or a computational budget is exhausted.

A task mapping solution is considered optimal if all jobs meet their deadlines at all levels of assurance (admissible $\mathcal{S}$) and the worst-case sub-frame lengths are minimized, implying a balanced workload distribution. Based on these requirements, we define the cost function of the optimization problem as:

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

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:

$$late(f,\ell) = \sum_{i=1}^{L} barriers(f,\ell)_i - \mathcal{L}_f. \quad (2)$$

If $late(f,\ell) > 0$, the tasks in f cannot complete execution by the end of the frame for their ℓ-level execution profiles. Therefore, with this cost function, we initially guide DSE towards finding an admissible solution. When such a solution is found, cost c_1 becomes negative or 0. Then, c_2, i.e., the 3rd norm of all sub-frame lengths, $\forall f \in \mathcal{F}, \forall \ell \in \{1,\ldots,L\}$, is used to minimize the worst-case lengths of all sub-frames.

Note that the $barriers$ function is computed for each visited solution. We estimate the WCRT of each task in every frame f by considering the worst-case delay that the concurrently executed tasks (in the same sub-frame) can cause to it. This delay, denoted as $d_i(f,\ell)$, depends on the memory mapping $\mathcal{M}_{mem}$, which defines which of the concurrently executed tasks are interfering with τ_i, and the bank arbitration policy. For bounded $d_i(f,\ell)$, the WCRT of τ_i in frame f at level ℓ is computed as:

$$WCRT_i(f,\ell) = e_i^{max}(\ell) + \mu_i^{max}(\ell) \cdot T_{acc} + d_i(f,\ell). \quad (3)$$

The task mapping optimization method can be easily extended to account for fixed task preemption points, dependencies among tasks with equal periods, mapping constraints, solution ranking, among others. Please refer to [2] for a more detailed discussion.

B. Memory Mapping ($\mathcal{M}_{mem}$) Optimization

The goal of memory mapping optimization is to determine a static allocation of the tasks' private data and communication buffers (memory blocks) BL to memory banks B of the shared DRAM ($\mathcal{E}_2$ of MIG $\mathcal{I}$), so that the timing interference of tasks when accessing the memory is minimized. Also, the total size of the allocated memory blocks in a bank should not surpass the bank capacity. This constraint holds for the memory mapping in Figure 2.

For this purpose, we adopt a heuristic method based on SA, similar to the task mapping optimization problem. The method is presented in the form of pseudocode in Listing 1. It receives as inputs an initial temperature T_0, a temperature decreasing factor $a \in (0,1)$, the maximum number of consecutive variations with no cost improvement that can be checked for a particular temperature $Fail_{max}$, a stopping criterion in terms of the final temperature T_{final}, and a stopping criterion in terms of search time (computational budget) $time_{max}$. It returns the best encountered solution(s) in the given time.

The algorithm starts with an arbitrary solution S, satisfying the bank capacity constraints. If **GenerateInitialSolution** can provide no such solution, exploration is aborted. Otherwise, DSE is performed by examining random variations of the memory mapping. Particularly, **Variate** selects arbitrarily a memory block and remaps it to a different memory bank such that no bank capacity constraint is violated. The new solution S' is accepted if $e^{-\left(Cost(S')-Cost(S)\right)/T}$ is no lesser than a randomly selected real value in (0,1). The cost of S' is, also, compared to the minimum observed cost, $Cost_{min}$. If it is lower than $Cost_{min}$, the new solution and its cost are stored, even if transition to S' was not admitted. The temperature T of SA is reduced geometrically with factor a. Reduction takes place every time a sequence of $Fail_{max}$ consecutive solutions are checked, none of which improves $Cost_{min}$. After temperature reduction, exploration continues from the so-far best found solution (S_{cur_best}). DSE terminates when the lowest temperature T_{final} is reached or the computational budget $time_{max}$ is exhausted.

Memory mapping affects the WCRT of a task τ_i by defining which of the tasks that can be concurrently executed to τ_i are interfering with it. The less interfering tasks, the lower the delay τ_i experiences when accessing the shared memory. Therefore, to evaluate a memory mapping solution we select a cost function that reflects the increase in task WCRT due to interference on the DRAM banks. We represent this in terms of a two-dimensional matrix D ($n \times n$), where $D_{i,j}$ describes the maximum delay task τ_i can suffer when executed concurrently with τ_j. $D_{i,j}$ is positive if τ_i and τ_j ($i \neq j$) are (i) of the same criticality level and (ii) interfering, i.e., accessing memory blocks in at least one common bank.

Algorithm 1 Modified SA for Memory Mapping $\mathcal{M}_{mem}$

```
Input: $T_0$, $a$, $Fail_{max}$, $T_{final}$, $time_{max}$
Output: $\bar{S}_{best}$
 1: $S \leftarrow$ GenerateInitialSolution()
 2: if $S == \emptyset$ then
 3:     return $null$
 4: end if
 5: $\bar{S}_{best} \leftarrow \{S\}$, $S_{cur\_best} \leftarrow S$, $Cost_{min} \leftarrow$ Cost($S$)
 6: $T \leftarrow T_0$
 7: $FailCount \leftarrow 0$
 8: $time \leftarrow$ StartTimer()
 9: while $time < time_{max}$ and $T > T_{final}$ do
10:     $S' \leftarrow$ Variate($S$)
11:     if $e^{-(\textbf{Cost}(S') - \textbf{Cost}(S))/T} \geq$ Random(0,1) then
12:         $S \leftarrow S'$
13:     end if
14:     UpdateBestSolutions($S'$)
15:     if Cost($S'$) $< Cost_{min}$ then
16:         $S_{cur\_best} \leftarrow S'$
17:         $Cost_{min} \leftarrow$ Cost($S'$)
18:         $FailCount \leftarrow 0$
19:     else
20:         $FailCount \leftarrow FailCount + 1$
21:     end if
22:     if $FailCount == Fail_{max}$ then
23:         $T \leftarrow a \cdot T$
24:         $S \leftarrow S_{cur\_best}$
25:         $FailCount \leftarrow 0$
26:     end if
27: end while
```

For the computation of D, two classes of bank arbitration policies are of particular interest. The class of **fair schedulers** are characterized by that an upper bound on the WCRT of a task can be derived independently of other concurrently executed tasks and their execution profiles. Here, an access request issued by a task can wait for at most a fixed number of access requests from other tasks before it is served. In this class of schedulers fall FCFS, RR, and TDMA. The other class of **work-conserving schedulers** is characterized by that an access issued by a task may have to wait for all concurrent tasks to finish issuing accesses before it is served. In this class fall fixed-priority policies. Here, we can still compute an upper WCRT bound by considering the execution profiles of the concurrent tasks. Note that FCFS and RR also fall in this class and hence, the upper bound computed here is also valid for them.

In particular, for the FCFS and RR policies, the required delay bound can be refined given that each access from τ_i can be delayed by at most one access from any other concurrently executed task to the same bank. That is because each core has at most one pending request at a time. This yields:

$$D_{i,j} = \sum_{\substack{b,bl:(\tau_i,bl)\in\mathcal{E}_1 \\ \wedge(bl,b)\in\mathcal{E}_2}} \sum_{\substack{bl':(\tau_j,bl')\in\mathcal{E}_1 \\ \wedge(bl',b)\in\mathcal{E}_2}} \min\{w((\tau_i,bl)), w((\tau_j,bl'))\}\cdot T_{acc}$$

As an example, matrices D are given below for a general work-conserving arbitration policy and for FCFS/RR, respectively, based on the MIG of Figure 2.

$$\left|\begin{array}{cccc} 0 & 20{\cdot}T_{acc} & 0 & 0 \\ 10{\cdot}T_{acc} & 0 & 10{\cdot}T_{acc} & 0 \\ 0 & 10{\cdot}T_{acc} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right| \quad \left|\begin{array}{cccc} 0 & 10{\cdot}T_{acc} & 0 & 0 \\ 10{\cdot}T_{acc} & 0 & 10{\cdot}T_{acc} & 0 \\ 0 & 10{\cdot}T_{acc} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right|$$

We assume that tasks τ_1, τ_2, τ_3 are of CL 2, whereas τ_4 has CL 1. D represents the worst-case mutual delays for the case when τ_1, τ_2, τ_3 can be executed in the same FTTS sub-frame. Note that the worst-case delay of each task τ_i can be derived from D as $\sum_{\tau_j\in\tau} D(i,j)$ if all tasks with CL χ_i would run in parallel. This information is used for WCRT analysis for a particular task mapping solution, see Eq. 3 (factor $d_i(f,\ell)$).

In the memory mapping optimization problem, the more distributed the memory blocks are across the banks, the less mutual delays will be incurred. One alternative to solve this optimization problem, is to compute (part of) the Pareto set of memory mapping solutions with minimal interference between any two tasks of the same CL. Algorithm 1 maintains such a set $\bar{S}_{best}$ of non-dominated solutions. In particular, a newly visited mapping solution S' with matrix D' is inserted to the set $\bar{S}_{best}$ if it has a lower value for at least one element of D' than the corresponding element of any solution in $\bar{S}_{best}$. If a solution $S \in \bar{S}_{best}$ is dominated by S', then S is removed from the set. This update is performed by **UpdateBestSolutions**.

Another alternative is to define a cost function D_{avg} as the average over all elements of matrix D, i.e., the average delay tasks of the same CL cause to each other when interfering on shared banks. Then we can find the best solution in terms of D_{avg}. In this case, $\bar{S}_{best}$ contains only one solution characterized by the minimum encountered D_{avg}.

C. Integration of Task and Memory Mapping Optimization

The problems of optimizing $\mathcal{M}_\tau + \mathcal{S}$ and $\mathcal{M}_{mem}$ are interdependent. Namely, DSE for the optimization of the task mapping requires information on the memory mapping for computing function $barriers$. Similarly, D_{avg}, i.e., the cost of a memory mapping solution, can be refined for a particular task mapping, depending on the tasks that can be executed in parallel. In the following, we outline two alternative approaches towards an integrated optimization solution.

I. **Task mapping optimization for each memory mapping in Pareto set** As discussed previously, one can compute using Algorithm 1 part of the Pareto set $\bar{S}_{best}$ of memory mapping solutions that minimize the interference between any two tasks of the same CL. These solutions consider that all tasks of the same CL are potentially executed in parallel (*worst-case task mapping*). The next step is to solve the task mapping optimization problem for each memory mapping in the set $\bar{S}_{best}$. Finally, the combination of solutions which minimizes $||barriers||_3$ is selected.

II. **Iterative task and memory mapping optimization** Since the complexity of computing the Pareto set solutions for the memory mapping optimization problem can be prohibitive, one can select an iterative solution to the two problems. Then, for each visited solution during DSE for $\mathcal{M}_\tau + \mathcal{S}$, a memory mapping optimization is also performed to find the solution with minimized cost $||barriers||_3$.

V. Experimental Evaluation

To evaluate the proposed design optimisation approaches, we use an industrial implementation of a flight management

Table I. MINIMIZED COST, ADMISSIBILITY AND OPTIMIZATION TIME FOR THE THREE DESIGN OPTIMIZATION ALTERNATIVES

	No mem. info	Integr. appr. I	Integr. appr. II
Min. $\|\|barriers\|\|_3$	893.8	545.2	545.2
Admissibility	FALSE	TRUE	TRUE
Initial $\mathcal{M}_{mem}$ opt.	-	5.8 sec	-
$\mathcal{M}_\tau + \mathcal{S}, \mathcal{M}_{mem}$ opt.	2 sec	60.5 min	8.2 min

system. It consists of 13 tasks (7 of CL 1, 6 of CL 2) for sensor reading, localization, and computation of the nearest airport. The periods of the tasks vary among 200 ms, 1 sec, and 5 sec. Their worst-case execution times were derived through measurements on a real system. For the level-2 profiles $C_i(2)$ of tasks τ_i with $\chi_i = 2$, we augment the worst observed execution times by a factor of 2. Similarly, for the memory accesses, we consider pessimistic bounds and derive the $C_i(2)$ parameters by multiplying these bounds by 1.5. For MIG $\mathcal{I}$, we model the following memory blocks: one block per task with size equal to the size of its data as measured on the deployed system, and one block per communication buffer with known size, too.

For a platform with 8 cores and a shared DRAM with 8 banks, each arbitrated according to a RR policy ($T_{acc} = 180\mu s$), we perform design optimization under the following settings: (i) optimization of $\mathcal{M}_\tau + \mathcal{S}$ by ignoring structure of the memory subsystem, (ii) optimization of $\mathcal{M}_\tau + \mathcal{S}$ and $\mathcal{M}_{mem}$ according to integrated approach I of Section IV-C, and (iii) similarly, according to integrated approach II. Note that for WCRT analysis in the first case, it is assumed that all tasks of the same CL are interfering. Therefore, each access of a task is delayed by all cores with at least one mapped task in the same FTTS sub-frame.

Table I compares the minimum encountered cost for the $\mathcal{M}_\tau + \mathcal{S}$ problem ($||barriers||_3$) as well as the time needed for optimization under the three settings. The SA algorithm was configured with parameters: $a = 0.9$, $Fail_{max} = 100$, T_0 based on the cost of 300 random solutions, $T_{final} = 0.1$, $time_{max} = 60$ min. For the optimization of $\mathcal{M}_\tau + \mathcal{S}$, the probabilities of selecting a sub-frame or core variation were 0.85 and 0.15, respectively. The FTTS cycle ($H = 5$ sec) was dimensioned with 25 frames (200 ms each).

Note that the minimized cost for the task mapping problem when memory mapping is accounted for (integrated approaches) is 38.3% lower than when it is ignored. This leads the first optimization alternative to fail in finding an admissible schedule for the considered task set, although admissible schedules exist when the memory blocks are distributed appropriately among the DRAM banks. For the particular case study, integrated approaches I and II result in the same optimized combination of task and memory mapping. The difference in time requirements stems from the different complexity of the two optimization problems. For integrated approach I, task mapping optimization was repeated for the 2461 memory mappings of the initially computed Pareto set. For integrated approach II, memory mapping optimization was respectively performed for the 9266 visited task mapping solutions. However, a memory mapping optimization in the second case converged much faster (avg. 50 ms) than a task mapping optimization in the first case (avg. 1.47 sec).

In general, depending on the sizes of the search spaces of the task mapping and memory mapping optimization problems, one algorithm can perform faster than the other. Regardless of which algorithm is chosen, results confirm that memory mapping optimization cannot be ignored as maximizing the slack time at the end of each FTTS frame is crucial for an even workload distribution and also, for enabling the scheduling of additional tasks that may be added later.

VI. CONCLUSION

This paper presents a design optimization method for mixed-criticality periodic task sets on multicores. Task mapping is optimized under the FTTS scheduling policy. At the same time, the tasks' private and shared data are mapped to memory banks to minimize the task interferences on the shared memory. Two alternative integrated solutions to the optimization problems are presented. Evaluation with an industrial application shows that accounting for and optimizing memory mapping can greatly reduce the task response times. This increases the probability of finding admissible scheduling solutions and enables efficient resource utilization.

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.

REFERENCES

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

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

[3] "Kalray mppa-256 manycore platform," http://www.kalray.eu/products/mppa-manycore/mppa-256/.

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

[5] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, pp. 1–36, 2013.

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

[7] R. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *ECRTS*, 2012, pp. 309–320.

[8] 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.

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

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

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

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

[13] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Operation and data mapping for cgras with multi-bank memory," in *LCTES*, 2010, pp. 17–26.

[14] W. Mi, X. Feng, J. Xue, and Y. Jia, "Software-hardware cooperative dram bank partitioning for chip multiprocessors," in *Network and Parallel Computing*, ser. LNCS, 2010, vol. 6289, pp. 329–343.

[15] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012, pp. 367–376.

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

[17] 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*, vol. 28, no. 7, pp. 966 –978, 2009.

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