Design Optimization of Mixed-Criticality Real-Time Embedded Systems

Domiţian Tămaş–Selicean, Technical University of Denmark
Paul Pop, Technical University of Denmark

In this paper we are interested in implementing mixed-criticality real-time embedded applications on a given heterogeneous distributed architecture. Applications have different criticality levels, captured by their Safety-Integrity Level (SIL), and are scheduled using static-cyclic scheduling. According to certification standards, mixed-criticality tasks can be integrated onto the same architecture only if there is enough spatial and temporal separation among them. We consider that the separation is provided by partitioning, such that applications run in separate partitions, and each partition is allocated several time slots on a processor. Tasks of different SILs can share a partition only if they are all elevated to the highest SIL among them. Such elevation leads to increased development costs, which increase dramatically with each SIL. Tasks of higher SILs can be decomposed into redundant structures of lower SIL tasks. We are interested to determine (i) the mapping of tasks to processors, (ii) the assignment of tasks to partitions, (iii) the decomposition of tasks into redundant lower SIL tasks, (iv) the sequence and size of the partition time slots on each processor, and (v) the schedule tables, such that all the applications are schedulable and the development costs are minimized. We have proposed a Tabu Search-based approach to solve this optimization problem. The proposed algorithm has been evaluated using several synthetic and real-life benchmarks.

Categories and Subject Descriptors: C.3 [**Special-Purpose and application-based systems**]: Real-time and embedded systems; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms: Embedded systems, Real-time systems, Mixed-criticality, metaheuristic, IMA

1. INTRODUCTION

Safety is a property of a system that will not endanger human life or the environment. *Safety-Integrity Levels* (SILs) are assigned to safety-related functions to capture the required level of risk reduction, and will dictate the development processes and certification procedures that have to be followed [IEC 61508 2010], [ISO 26262 2009], [RTCA DO-178B 1992]. There are multiple SIL levels in safety standards, e.g., IEC 61508 [2010] has four SIL levels, ranging from SIL 4 (most critical) to SIL 1 (least critical). Certification standards require that safety functions of different criticality levels are *protected* (or, *isolated*), so they cannot influence each other. For example, without protection, a lower-criticality task could corrupt the memory of a higher-criticality task.

The "Research Agenda for Mixed-Criticality Systems" [Barhorst et al. 2009] defines a *mixed-criticality* system as "an integrated suite of hardware, operating system and middleware services and application software that supports the execution of safety-critical, mission-critical, and non-critical software within a single, secure computing platform". Many such applications, following physical, modularity or safety constraints, are implemented using distributed architectures, composed of several different types of hardware components (called *nodes*), interconnected in a network. Initially, each function was implemented in a separate node, which has led to a large increase in the number of nodes. The current trends are towards "integrated architectures", where several

This work has been funded by the Advanced Research & Technology for Embedded Intelligence and Systems (ARTEMIS) within the project 'RECOMP', support code 01IS10001A, agreement no. 100202.
Author's addresses: D. Tămaş–Selicean and Paul Pop, DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark.

DOI:http://dx.doi.org/10.1145/0000000.0000000

functions are integrated onto the same node. In this context, designers are relying on partitioning mechanisms at the platform level. Without the separation provided by partitioning, all the functions sharing the same resource would have to be certified at the highest SIL among them, which would be prohibitively expensive (the certification costs can increase the costs of the project anywhere from 25 to 100% [IBM 2010]). In the avionics area, the partitioned architecture is called "Integrated Modular Avionics" (IMA) [ARINC 1997], and the platform-level separation mechanisms are provided by implementations of the ARINC 653 standard [ARINC 2013]. ARINC 653 consists of hardware-mediated operating system-level *spatial* and *temporal* partitioning mechanisms [Rushby 1999]. Similar platform-level separation mechanisms are available in other industries [Ernst 2010; Leiner et al. 2007; **?**].

In this paper we are interested in the design optimization of hard real-time applications of different SILs. We consider distributed platforms, consisting of several processing elements (PEs) interconnected using a broadcast bus. We assume each PE contains a CPU, RAM and non-volatile memory, and a network interface card. We assume that the platform provides both spatial and temporal partitioning, thus enforcing enough separation for the mixed-criticality applications. Each partition can have its own scheduling policy. There has been a long debate in the real-time and embedded systems communities concerning the advantages of using a Time-Triggered (TT) or Event-Triggered (ET) approach [Audsley et al. 1993; Kopetz 2011a; Xu and Parnas 1993]. Several aspects have been considered in favor of one or the other approach, such as flexibility, predictability, jitter control, processor utilization, testability, etc. Our work can handle both TT tasks scheduled using static-cyclic scheduling (SCS) and ET tasks scheduled using fixed-priority pre-emptive scheduling (FPS). In [Tămaş-Selicean and Pop 2011] we have shown how applications scheduled using a FPS policy can be handled in a partitioned architecture. However, to simplify the discussion, in this paper, we assume that all applications are scheduled using SCS, which is the preferred scheduling policy for highly critical systems. In addition, although we address hard real-time applications, (non-critical) soft real-time applications can also be handled using a technique such as the Constant Bandwidth Server [Abeni and Buttazzo 1998], where the server is seen as a hard task providing a desired level of service to soft tasks.

We assume that the communication protocol has mechanisms to enforce partitioning at the bus level. For example, space partitioning is attained in SAFEbus [Hoyme and Driscoll 1993] by mapping the messages to unique locations in the inter-module memory, protected by a memory-mapping hardware in the host, and temporal partitioning is achieved in TTP [Kopetz 2011b] by enforcing a Time-Division Multiple Access scheme. TTEthernet [AS 6802 2011] offers spatial separation for mixed-criticality messages through the concept of virtual links, and temporal separation, enforced through schedule tables for time-triggered messages and bandwidth allocation for rate constrained messages. Researchers have shown how realistic bus protocols such as TTP [Pop et al. 2004], FlexRay [Pop et al. 2008b] and TTEthernet [Tămaş-Selicean et al. 2012] can be taken into account during the design. However, in this paper we consider a simple statically scheduled bus.

Safety-critical real-time applications have to function correctly and meet their timing constraints even in the presence of faults. Fault tolerance can be addressed with hardware architecture solutions, such as TTA [Kopetz 2011b], or software-based solutions such as re-execution, replication and checkpointing [Pop et al. 2009]. In this paper we do not address the issue of fault-tolerance (which is orthogonal to our problem), and we assume that the designer has developed the applications such that they provide the required level of fault-tolerance.

1.1. Contribution

In this paper we are interested in implementing mixed-criticality embedded real-time applications on a given distributed architecture, such that all applications are schedulable and the development costs are minimized. An implementation consists of (i) the mapping of tasks to PEs, (ii) the assignment of tasks to partitions, (iii) the decomposition of higher SIL tasks into redundant structures of lower SIL tasks, (iv) the sequence and size of the partition time slots on each PE, and (v) the schedule tables for all PEs. We propose a Tabu Search (TS)-based approach for this design optimization problem.

In [Tămaş-Selicean and Pop 2011] we have presented a Simulated Annealing-based approach for the optimization of the sequence and size of time slots, considering a given fixed mapping. As the experimental results will show, significant improvements can be obtained if mapping is considered at the same time with partitioning, as we propose in this paper. There are cases when obtaining schedulable implementations is not possible, even if mapping is considered at the same time with partitioning. In such cases, one option is to upgrade the hardware platform. This will increase the *unit cost* of the system. However, there are many cost-sensitive areas (e.g., automotive, which is a mass market), where increasing unit costs are not an option. Therefore, in this paper we address the case when the *sharing* of partitions by tasks from applications with different SILs is allowed, aiming at integrating more applications onto a given platform, without increasing unit costs. We also consider the decomposition of higher SIL tasks into lower SIL tasks, as allowed by certification standards. If tasks of different SILs share a partition, they will have to be developed and certified at the highest SIL level among them. This will increase the *development costs*. Thus, we integrate more applications onto the same cost-sensitive partitioned architecture at the expense of increased development costs, instead of increased unit costs.

The paper is organized in eight sections. The next two sections present the application and system models considered, respectively. The problem formulation is presented in Section 4. Our proposed TS optimization approach is outlined in Section 5 and evaluated in Section 6. The related work is presented in Section 7. The last section presents our conclusions.

2. APPLICATION MODEL

The set of all applications in the system is denoted by Γ. We model an application as a directed, acyclic graph $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \Gamma$. Each node $\tau_j \in \mathcal{V}_i$ represents one task. The mapping is denoted by the function $M : \mathcal{V}_i \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of processing elements (PEs) in the architecture. This mapping is not yet known and will be decided by our approach. For each task τ_i we know the worst-case execution time (WCET) $C_i^{N_j}$ on each processing element N_j where τ_i is considered for mapping.

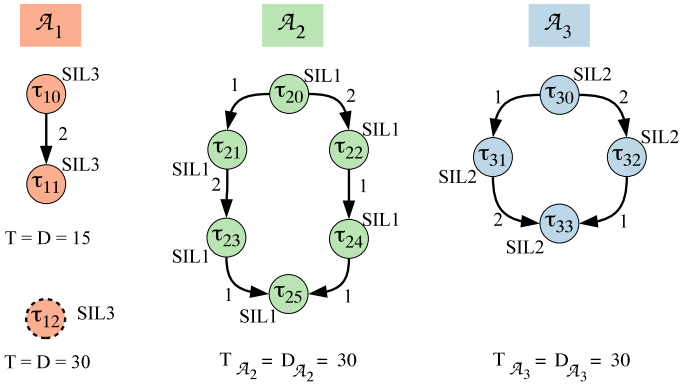
An edge $e_{jk} \in \mathcal{E}_i$ from τ_j to τ_k indicates that the output of τ_j is the input of τ_k. A task becomes ready after all its inputs have arrived, and it issues its outputs when it terminates. Communication between tasks mapped to different PEs is performed by message passing over the bus. The time necessary to transmit message m_i on the bus is captured by s_{m_i}, and is given. All the applications are scheduled using SCS. A deadline $D_{\mathcal{G}_i} \leq T_{\mathcal{G}_i}$, where $T_{\mathcal{G}_i}$ is the period of $\mathcal{G}_i$, is imposed on each application graph $\mathcal{G}_i$.

An example mixed-criticality system composed of three applications is presented in Fig. 1a. The periods and deadlines are presented under the application graphs. The WCETs of tasks are given in Fig. 1b for two PEs, N_1 and N_2. "N/A" in the table means that the task is not considered for mapping on the respective PE. The size of the messages is depicted on the graph edges.

During the engineering of a safety-critical system, the hazards are identified and their severity is analyzed, the risks are assessed and the appropriate risk control measures are introduced to reduce the risk to an acceptable level. A Safety-Integrity Level (SIL) captures the required level of risk reduction, see Section 7 for details. We introduce the function $SIL : \mathcal{V}_i \rightarrow \{SIL\,k\}$, where $k \in \{0..4\}$, to capture the SIL of a task. The tasks of an application may have different SILs. The SILs for the example in Fig. 1a are presented next to the tasks.

2.1. Task Decomposition

During the early stages of the design of safety-critical systems, a SIL is allocated to each safety function. Safety functions are later implemented as software or hardware, or a combination of both. Let us consider a safety function of SIL i, to be implemented as software tasks. The certification standards allow several options. For example, the safety-function could be implemented as one task of SIL i or, using redundancy to increase dependability, as several redundant tasks of a lower SIL, e.g., SIL i-1. Decomposing a safety function of a higher SIL into several redundant tasks



(a) Example mixed-criticality applications

	$\mathcal{A}_1$			$\mathcal{A}_2$						$\mathcal{A}_3$			
	τ_{10}	τ_{11}	τ_{12}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}	τ_{30}	τ_{31}	τ_{32}	τ_{33}
C^{N_1}	N/A	3	N/A	2	3	N/A	2	6	2	4	6	4	N/A
C^{N_2}	4	5	3	3	N/A	6	3	9	6	9	10	5	4

(b) WCET and mapping restrictions

	$\mathcal{A}_1$			$\mathcal{A}_2$						$\mathcal{A}_3$			
	τ_{10}	τ_{11}	τ_{12}	τ_{21}	τ_{21}	τ_{22}	τ_{23}	τ_{24}	τ_{25}	τ_{30}	τ_{31}	τ_{32}	τ_{33}
SIL 1	N/A	N/A	N/A	2	3	3	4	3	4	N/A	N/A	N/A	N/A
SIL 2	N/A	N/A	N/A	4	5	4	8	7	7	5	5	8	9
SIL 3	13	14	12	9	8	8	11	13	12	11	9	15	15
SIL 4	29	20	21	16	14	15	19	22	23	20	18	25	26

(c) Development costs (kEuro)

Fig. 1: Application model example

of lower SILs can reduce the development and certification costs, and could be the right choice in a particular context. For software redundancy, the standards recommend the use diversity, i.e., different implementations of the same functionality. This is because a fault (bug) in a software task will lead to a correlated failure in all of the tasks sharing the same implementation, unless software diversity is used. Often, one of the redundant tasks will implement a simpler (and maybe less accurate) algorithm as alternative diverse implementation.

Certification standards refer to this process as "SIL decomposition" and provide recommendations on the possible decompositions. For example, ISO 26262[1], Part 9, Section 5, provides the guide shown in Table I for SIL decomposition. Such a decomposition guide amounts to a "SIL algebra" [Parker et al. 2013], i.e., the SIL of the safety function is the sum of the SILs of the redundant tasks.

In this paper we assume that the safety functions are implemented as software tasks running on a distributed architecture. Let us consider a tasks τ_A which has to fulfill a safety requirement of SIL 3. According to Table I, we can decompose task τ_A into two redundant tasks, e.g., τ_B with SIL 2 and τ_C of SIL 1. Task τ_B can be further decomposed into two SIL 1 tasks.

[1]ISO 26262 uses the concept of Automotive SIL, or ASIL. To simplify the discussion, we consider ASIL D to be SIL 4 and ASIL A to be SIL 1.

Table I: ISO/DIS 26262 SIL decomposition schemes

SIL	Can be decomposed as
SIL 4	SIL 3 + SIL 1 or SIL 2 + SIL 2 or SIL 4
SIL 3	SIL 2 + SIL 1 or SIL 3
SIL 2	SIL 1 + SIL 1 or SIL 2
SIL 1	SIL 1

We assume that, for those tasks which are considered for decomposition, the designer will specify a library $\mathcal{L}$ of possible decompositions based on the standard considered, similar to the library in Table I. In this paper, we are interested to determine a decomposition of tasks such that the timing requirements are satisfied and the development costs are minimized.

Fig. 5a shows a two decomposition options for task τ_{11} of SIL 4 from application $\mathcal{A}_1$ in Fig. 4a. We define the decomposition function $D(\tau_i)$, $D(\tau_i) : \mathcal{V}_i \rightarrow \mathcal{D}_i$, where $\mathcal{D}_i$ is a set of decomposition options, specified in the decomposition library $\mathcal{L}$. There are two decomposition options in Fig. 5a: D_1 into tasks τ_{11b} of SIL 3 and τ_{11c} of SIL 1, and D_2 in two tasks of SIL 2, i.e., τ_{11f} and τ_{11g}. Fig. 5a also shows how τ_{11}, once decomposed, is connected to the graph of application $\mathcal{A}_1$ from Fig. 4a. We assume that a decomposed task will be connected to the original application graph via two "connecting" tasks; one task which is distributing the input to the redundant decomposed tasks (τ_{11a} in Fig. 5a) and one task which is collecting the outputs (τ_{11d}). The SIL of the connecting tasks are given by the engineer based on the requirements from the standards.

2.2. Development Cost Model

The SIL assigned to a task will dictate the development processes and certification procedures that have to be followed. Software development cost estimation is a widely researched topic, and is beyond the scope of this paper. The reader is directed to [Jorgensen and Shepperd 2007; Boehm et al. 2000a] for reviews on this topic. One of the most influential software cost models is the Constructive Cost Model (COCOMO) [Boehm et al. 2000b]. Researchers have shown how to take into account the development costs during the design process of embedded systems [Debardelaben et al. 1997].

The development of safety-critical systems is a highly structured and systematic process dictated by standards. These standards increase the development costs due to additional processes for software development and testing, qualification activities involved in compliance and increased process complexity, shown also by an IBM Rational study [IBM 2010]. Because of the systematic nature of the development processes dictated by the standards, we assume that the designer will be able to estimate the development effort required for a task. Hence, we define the development cost (DC) function $DC(\tau_i, SIL\ j)$ to capture the cost to develop and certify a task τ_i to safety integrity level *SIL j*. Fig. 1c shows an example of the development costs for each of the tasks in Fig. 1a. Knowing the DC for each task, we can compute this cost at the application level. The DC of application $\mathcal{A}_i$, denoted by $DC(\mathcal{A}_i)$, is the sum of the development costs of each task in the application. Similarly, we define the DC for the set of all the applications, $DC(\Gamma)$, as the sum of the costs for each application. An example certification cost estimation in person-days for an Air Traffic Control radio platform is presented in [Rockwell-Collins 2009].

3. SYSTEM MODEL

We consider architectures composed of a set $\mathcal{N}$ of PEs which share a broadcast communication channel. In this paper we focus on minimizing development costs, and not on the communication, hence we consider a simple statically scheduled bus, where the communication takes place according to a static schedule table computed offline. Also, we consider that all applications are scheduled using non-preemptive static-cyclic scheduling (SCS).

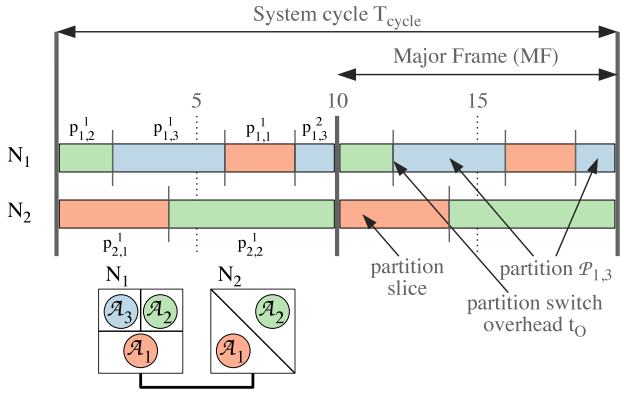


Fig. 2: Partitioned architecture

3.1. Protection through Partitioning

When several tasks of different SILs share the same processing element, the standards require that they are developed at the highest SIL among the SILs of the tasks, which is very expensive. Unless, the standards state, it can be shown that the implementation of the tasks is "sufficiently independent", i.e., there is both spatial and temporal separation among the tasks. Hence, tasks of different SILs have to be protected from each other. Otherwise, for example, a lower-criticality task could interfere with the activity of a higher-criticality task, leading thus to a failure. Protection also imposes constraints on the type of communication that is allowed. Thus, within an application, a task can only receive an input from a task of the same criticality level or higher than its own. In addition, we assume that there is no communication between two applications.

We consider that the protection is achieved through a temporal- and space-partitioning scheme similar to Integrated Modular Avionics (IMA) [Rushby 1999]. Partitioning schemes similar to IMA are available in several application areas [Pop et al. 2013], not only in the avionics area. Space partitioning uses mechanisms such as a Memory Management Unit (MMU) to ensure that, for example, applications running on different partitions cannot corrupt the memory for the other applications. Temporal partitioning ensures the access of each application to the CPU, according to a predetermined partition table. A detailed discussion about partitioning is available in [Rushby 1999].

We denote the assignment of tasks to partitions using the function $\phi : \mathcal{V} \rightarrow \mathcal{P}$, where $\mathcal{V}$ is the set of tasks in the system and $\mathcal{P}$ is the set of partitions. On a processing element N_i, a partition $P_j \in \mathcal{P}$ is defined as the sequence $\mathcal{P}_{i,j}$ of k partition slices $p^k_{i,j}$, $k \geq 1$. A partition slice $p^k_{i,j}$ is a predetermined time interval in which the tasks mapped to N_i and allocated to the partition P_j are allowed to use N_i.

All the slices on a processor are grouped within a time interval called "Major Frame" (MF), that is repeated periodically. The period T_{MF} of the Major Frame is given by the designer and is the same on each PE. Several MFs are combined together in a system cycle that is repeated periodically, with a period T_{cycle}. Within a T_{cycle}, the sequence and length of the partition slices are the same across MFs (on a given PE), but the contents of the slices (i.e., the tasks assigned to the slices) can differ. In this work we have considered that the values for T_{MF} and the T_{Cycle} are given, since in real-life systems these are difficult to change because of legacy applications. In case such a change is permitted, we have shown in [Tamas-Selicean et al. 2014] how the strategy proposed in this paper can be extended to also determine the values for T_{MF} and T_{Cycle}.

Fig. 2 presents the partitions for 3 applications of different SILs, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$, implemented on an architecture of 2 PEs, N_1 and N_2, with $T_{MF} = 10$ and $T_{cycle} = 2 \times T_{MF} = 20$. Using the partitions in the figure, the tasks of $\mathcal{A}_3$, for example, can execute only in partition P_3 on PE N_1 (the light blue rectangles in the timeline on N_1), composed of the sequence $\mathcal{P}_{1,3}$ of partition slices $p^1_{1,3}$ and $p^2_{1,3}$. In this example, we assume that all the tasks of $\mathcal{A}_3$ have the same SIL. However, as mentioned

in Section 2, the tasks of an application may have different SILs. Such tasks have to be placed in separate partitions.

The schedule tables $\mathcal{S}$ for the applications have to be constructed such that they take into account the partitions $\mathcal{P}$. Note that a task can extend its execution over several partition slices and MFs. When a task does not complete during a partition slice, its execution is suspended until its partition is activated again. Such an example is task τ_{31} on N_1 in Fig. 3b, which shows the schedule tables for the applications in Fig.1a. The time overhead due to partition switching is denoted by t_O, and our optimization approach takes into account the partition switching overheads.

3.2. Elevation and Software-based Protection

As mentioned, tasks of different SILs have to be placed in separate partitions. However, there might be situations when it would be beneficial (e.g., in terms of schedulability) for two tasks of different SILs to share a partition. This can be achieved through *elevation*: increasing the SIL of the lower criticality task to the level of the higher criticality task. Although such elevation to a higher SIL is allowed by the standards, it will increase the development costs for the elevated task. For example, considering the application details from Fig. 1, in Fig. 3d task τ_{23} shares the partition with tasks τ_{12} on N_2, second MF. As task τ_{23} has a lower SIL than τ_{12}, which is SIL 3, it has to be elevated from SIL 1 to SIL 3. This is shown visually in the schedule by raising task τ_{23} slightly compared to the other tasks which are not elevated.

Such elevation may trigger the elevation of other tasks. For example, as previously mentioned, we assume that a task can only receive inputs from predecessors of the same or higher SIL. This means that elevating a task τ_i to a higher SIL may trigger the elevation of its predecessors, triggering the elevation of other tasks, if such predecessors will be assigned a higher SIL in another partition slice. This is the case for the tasks in application $\mathcal{A}_2$, shown in Fig. 3d using green rectangles. As τ_{22} and τ_{23} share the partition with τ_{12}, they have to be elevated from SIL 1 to SIL 3. This in turn triggers the elevation to SIL 3 of the predecessors of these tasks, namely tasks τ_{20} and τ_{21}, see the $\mathcal{A}_2$ graph in Fig. 1a. Furthermore, τ_{20} and τ_{21} share the partition with τ_{24} and τ_{25}, namely partition $\mathcal{P}_2$ on N_1, see Fig. 3d. Since tasks τ_{20} and τ_{21} were elevated to SIL 3, and tasks τ_{24} and τ_{25} are SIL 1, we need to elevate tasks τ_{24} and τ_{25} to SIL 3 to allow the sharing of partition $\mathcal{P}_2$. Thus, all the tasks of application $\mathcal{A}_2$ are elevated to SIL 3, increasing the development costs DC for this application from 19 to 61 kEuro (the development costs for these tasks are presented in Fig. 1c).

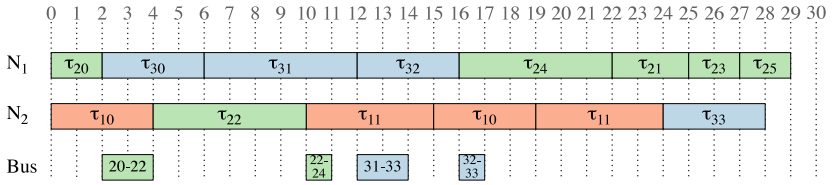
4. PROBLEM FORMULATION

The problem we are addressing in this paper can be formulated as follows: given a set Γ of applications, the criticality level $SIL(\tau_i)$ of each task τ_i, the library of SIL decompositions $\mathcal{L}$, an architecture consisting of a set $\mathcal{N}$ of processing elements, the size of the major frame T_{MF} and the application cycle T_{cycle}, we are interested to find an implementation Ψ such that all applications meet their deadlines and the development costs are minimized. Deriving an implementation Ψ means deciding on (1) the SIL decomposition D of the tasks for which the designer has provided alternatives in the library $\mathcal{L}$, (2) the mapping M of tasks to PEs taking into account the mapping restrictions, (3) the set $\mathcal{P}$ of partition slices on each processor, including their order and size, (4) the assignment ϕ of tasks to partitions and (5) the schedule $\mathcal{S}$ for all the tasks and messages in the system.

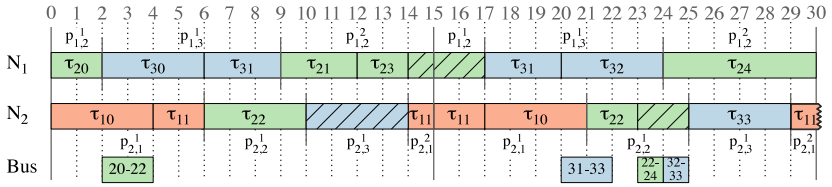
4.1. Partition-Aware Mapping Optimization

Let us illustrate the problem using the mixed-criticality applications $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ from Fig. 1a, to be implemented on two PEs, N_1 and N_2. We initially do not consider task τ_{12}, i.e., it is not part of application $\mathcal{A}_1$. We have set T_{MF} to 15 time units and $T_{cycle} = 2 \times T_{MF} = 30$. In this example we ignore the partition switch overhead. Note that in this subsection we do not yet consider partition sharing by tasks of different criticality, which is discussed in Section 4.2, nor do we consider SIL decompositioning, which is discussed in Section 4.3.

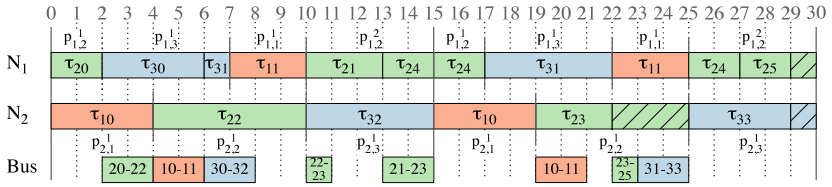
Let us first consider the case when the mapping and partitioning optimizations are performed separately. Thus, Fig. 3a presents the mapping and schedules for the case when there is no partition-



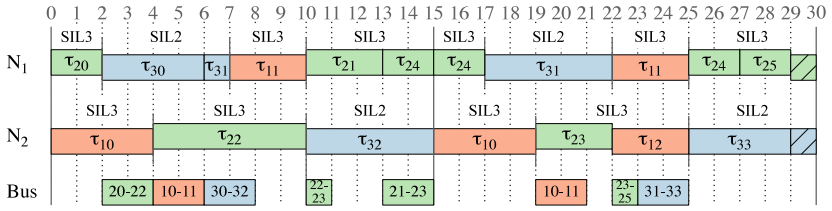
(a) Optimal mapping and schedules, without considering partitions



(b) Partitioning, using the previously obtained mapping. τ_{25} and the second instance of τ_{11} do not fit in the schedule



(c) By remapping tasks τ_{11}, τ_{23} and τ_{32}, and by optimizing the time partitions we manage to successfully schedule all the applications



(d) By elevating τ_{22} and τ_{23} to SIL 3, and thus all the tasks in $\mathcal{A}_2$, we manage to successfully schedule all applications

Fig. 3: Motivational example

ing, i.e., the tasks do not have to be separated, and they can use the PEs without restrictions. The mapping and scheduling are optimal in terms of schedulability, captured by the "degree of schedulability" metric, which is the sum of the slacks available between the completion time R_i of an application graph $\mathcal{A}_i$ and its deadline D_i. The "degree of schedulability" cost function is presented in Eq. 2 in Section 6. In Fig. 3a we show the schedules on each resource, namely, the PEs N_1 and N_2 and the bus, using a Gantt chart. The messages on the bus are labeled with the indices of the sender and receiver task, e.g., the first message on the bus, "20–22" is sent from task τ_{20} to τ_{22}. The dashed vertical lines are timeline guides to help with the visualization of the schedule, and should not be interpreted as partitions, since we ignore partitions in Fig. 3a.

Next, using this optimal mapping, we are interested to obtain the partitions and the schedules, such that, the separations are enforced and the schedule lengths are minimized with the goal of producing a schedulable implementation. Thus, Fig. 3b presents the optimal partitions and schedules (in terms of the same cost function from Eq. 2), considering the fixed mapping decided in Fig. 3a.

The continuous line at time 15 represents the major frame boundary, while the shorter continuous lines, such as the one between tasks τ_{20} and τ_{30} represent partition slice boundaries. The partition slices are denoted by the notation p_{ij}^{k} introduced in Section 3.1. We mark the unused CPU time of a partition slice with a hatching pattern, as is the case with partition slice p_{12}^{1} on N_1 in the second MF assigned to $\mathcal{A}_2$.

With partitioning, tasks can only execute in their assigned partition. Hence, partitioning may lead to unused slack in the schedule, even in the case of an optimal partitioning and schedule, as depicted in Fig. 3b. In this case, although application $\mathcal{A}_3$ is schedulable, task τ_{25} of $\mathcal{A}_2$ does not fit into the schedule. Furthermore, given that task τ_{11} has only executed for 2 time units in slot 4–6 and 1 time unit in slot 14–15 when it reaches its deadline, there are still 2 more time units that must be executed, hence τ_{11} misses the deadline. Thus applications $\mathcal{A}_1$ and $\mathcal{A}_2$ are not schedulable.

Our approach in this paper is to perform the optimization of mapping and partitioning at the same time, and not separately. By deciding simultaneously the mapping and partitioning we have a better chance of obtaining schedulable implementations. Such a solution is depicted in Fig. 3c, where all applications are schedulable. Compared to the solution in Fig. 3b, we have changed the mapping of tasks τ_{23} and τ_{32} from N_1 to N_2 and of task τ_{11} from N_2 to N_1, and we have resized the partition slices and changed the schedule accordingly. This example shows that by optimizing the mapping at the same time with partitioning we are able to obtain schedulable implementations.

4.2. Partition-Sharing Optimization

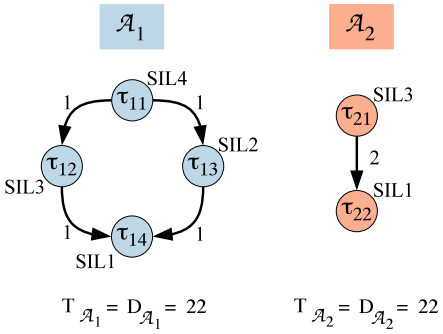
However, there might be cases when obtaining schedulable implementations is not possible, even if mapping and partitioning are considered simultaneously. For example, let us consider a similar setup as in the previous section, with the only difference that we add task τ_{12} to application $\mathcal{A}_1$, see Fig. 1a. The solution presented in Fig. 3c, prior to adding task τ_{12}, has the partitioning and scheduling optimized, and the schedule is almost full. Adding task τ_{12} to the task set, we are unable to obtain a schedulable implementation: although it may seem that task τ_{12} would fit in-between tasks τ_{23} and τ_{33} in the schedule of N_2 in Fig. 3c, τ_{12}, which is SIL 3, cannot use that partition, which is for SIL 1 tasks. Moreover, the partition slice $p_{2,2}^{1}$ cannot be split, because then τ_{22} would not fit in the first major frame.

For such situations, in this paper we consider the elevation of tasks to allow partition sharing, and we are interested to derive schedulable implementations that minimize the development costs associated to elevation. Thus, in Fig. 3d we allow τ_{12} of SIL 3 to share the partition with tasks τ_{22} and τ_{23} of SIL 1, by elevating these two tasks to SIL 3. This will trigger the elevation of the predecessors of τ_{22} and τ_{23}, namely τ_{20} and τ_{21}, to SIL 3. In addition, since τ_{20} and τ_{21} share partitions with tasks τ_{24} and τ_{25}, these will also have to be elevated to SIL 3, leading to a complete elevation of application $\mathcal{A}_2$ from SIL 1 to SIL 3, which, according to the costs from Fig. 1c, means an increase in development costs from 85 kEuros to 127 kEuros. The solution in Fig. 3d is schedulable, and is optimal in terms of development costs as captured by the cost function from Eq. 1 to be discussed in Section 5.1.

Note that, in many application areas, such a development cost increase is preferred to an increase in unit costs. Our optimization approach provides to a trade-off analysis tool to the designer, who can decide what is the best option: to upgrade the platform and increase the unit costs, or to increase the development costs, but keep the same architecture.

4.3. Task Decomposition

We have not yet discussed the issue of SIL decomposition. In the previous subsection we have shown how to use elevation to achieve partition sharing, which may lead to increased development costs. Another option is to explore several SIL decompositions for those tasks for which the designer has specified a SIL decomposition in the decomposition library $\mathcal{L}$ (see Section 2.1). Using SIL decomposition will result in more (redundant) tasks of lower SILs. Using multiple tasks of lower SILs has the advantage of lowering the development costs and may facilitate partition sharing.



(a) Two applications

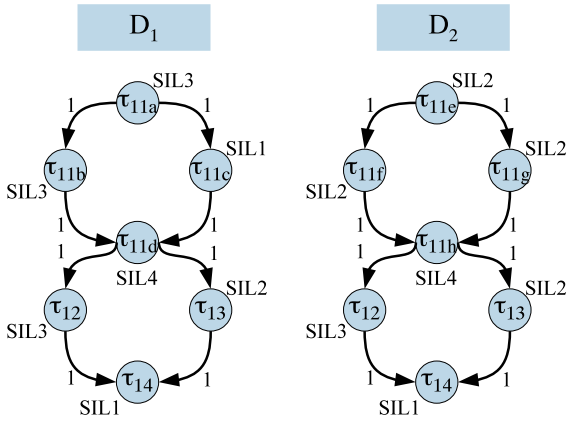
	$\mathcal{A}_1$				$\mathcal{A}_2$	
	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{21}	τ_{22}
C^{N_1}	5	5	4	3	5	3
C^{N_2}	4	7	6	4	6	4

(b) WCETs

	$\mathcal{A}_1$				$\mathcal{A}_2$	
	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{21}	τ_{22}
SIL 1	N/A	N/A	N/A	10	N/A	9
SIL 2	N/A	N/A	18	20	N/A	16
SIL 3	N/A	34	37	37	32	31
SIL 4	60	70	70	70	68	62

(c) Development costs (kEuro)

Fig. 4: Application model example for SIL decomposition



(a) Library $\mathcal{L}$ with two decompositions

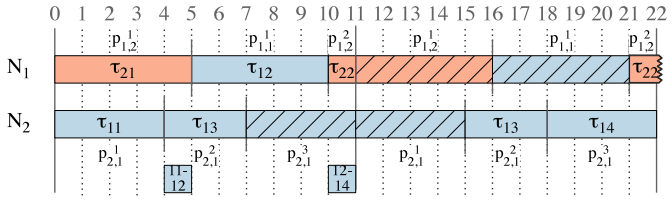
	τ_{11a}	τ_{11b}	τ_{11c}	τ_{11d}	τ_{11e}	τ_{11f}	τ_{11g}	τ_{11h}
C^{N_1}	1	5	1	2	1	3	3	2
C^{N_2}	1	4	1	2	1	2	2	2

(b) WCETs for the tasks resulted from the SIL decomposition

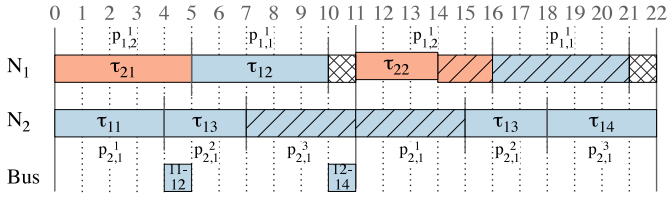
	τ_{11a}	τ_{11b}	τ_{11c}	τ_{11d}	τ_{11e}	τ_{11f}	τ_{11g}	τ_{11h}
SIL 1	N/A	N/A	9	N/A	N/A	N/A	N/A	N/A
SIL 2	N/A	N/A	18	N/A	N/A	14	13	N/A
SIL 3	1	33	32	N/A	1	30	29	N/A
SIL 4	1	60	60	10	1	60	60	10

(c) Developments costs for the tasks resulted from SIL decomposition

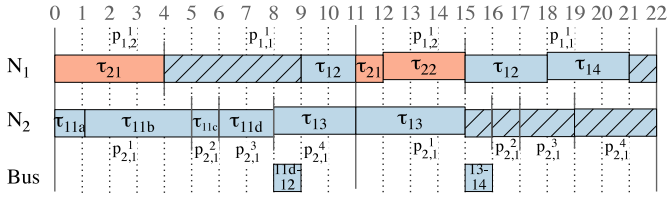
Fig. 5: Example decomposition for task τ_{11}



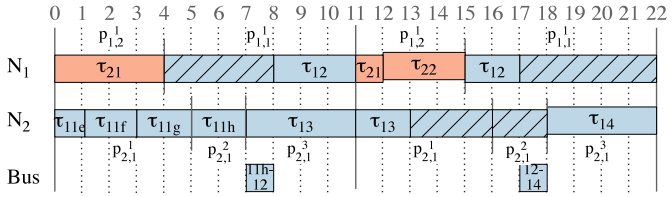
(a) No sharing or decomposition, τ_{22} does not fit into the schedule. Initial DC=172 kEuro



(b) By allowing partition sharing, we obtain a schedulable solution. DC=194 kEuro



(c) If not selected carefully, SIL decomposition may increase costs. DC=224 kEuro



(d) Using an optimized SIL decomposition can lower the development costs. DC=173 kEuro

Fig. 6: SIL decomposition optimization example

The disadvantage is the introduction of more tasks, which have to be placed in the schedule table, potentially impairing schedulability.

Let us illustrate these issues using the example in Fig. 4. We have two applications, $\mathcal{A}_1$ and $\mathcal{A}_2$, presented in Fig. 4a. The WCETs for the tasks are shown in Fig. 4b, while Fig. 4c shows the development costs. Let us assume the designer is considering decomposing task τ_{11} into two options D_1 and D_2 as discussed in Section 2.1 and shown in Fig. 5a. Fig. 5b and Fig. 5c present the WCETs and development costs of the tasks resulted from the decomposition. The T_{MF} is 11 time units, and T_{cycle} is 22.

We first show a solution to this example without considering partition sharing and SIL decomposition. Thus, Fig. 6a presents the optimal mapping, partitioning and schedules of tasks, as obtained by running the simultaneous mapping and partitioning optimization discussed in Section 4.1. In this case, τ_{22} does not fit into the schedule. Although $\mathcal{A}_2$ has two partition slices on N_1, i.e., $p^1_{1,2}$ and $p^2_{1,2}$, with a total time of 6 out of the 11 time units of the MF, τ_{22}, which is of SIL 1, is allowed to execute

only in $p^2_{1,2}$, since it cannot share the partition slice $p^1_{1,2}$ with τ_{21} of SIL 3. Thus, τ_{22} executes for only two time units, during time slot 10–11 and time slot 21–22, but there is still 1 more time unit left to execute by the time it reaches the deadline. Fig. 6b shows a solution where we allow partition sharing, but not SIL decomposition. In this case, a schedulable solution was obtained by elevating τ_{22} to SIL 3 to allow τ_{21} and τ_{22} to share the same partition slice $p^1_{1,2}$. Due to the elevation of τ_{22}, the development cost of this solution increased to 194 kEuros, compared to 172 kEuros, if all the tasks would have been implemented and certified according to their lowest possible SIL.

We show in Fig. 6c the solution when we use SIL decomposition alongside with partition sharing. In Fig. 6c we decompose τ_{11} of SIL 4 into two tasks of SIL 3 and SIL 1 as specified by the decomposition option D_1, see Fig. 5a. Decomposing τ_{11} in this manner increases the cost from 194 kEuros, corresponding to the solution in Fig. 6b, to 224 kEuros. To obtain a schedulable solution task τ_{13} is elevated from SIL 2 to SIL 3 to share the partition slice $p^1_{2,1}$ with the other tasks of SIL 3. Similarly, τ_{14} is elevated to SIL 3 to share $p^1_{1,1}$ with τ_{12}. Clearly, this decomposition does not help our design, as it significantly increases the costs. Hence, not all decompositions are improving the design. Fig. 6d presents a solution where we use the SIL decomposition specified by D_2, Fig. 5a. Thus, τ_{11} of SIL 4 is decomposed into two tasks of SIL 2, namely τ_{11f} and τ_{11g}. Similar to Fig. 6b, task τ_{22} is elevated to SIL 3 to share the partition slice $p^1_{1,2}$ with τ_{21}. Moreover, task τ_{14} is elevated to SIL 2 to share the partition slice $p^4_{2,1}$ with τ_{13}. Decomposing in this manner reduces the cost to 173 kEuros, while also ensuring that all deadlines are satisfied.

This example shows that, in order to reduce costs and obtain schedulable solutions, it is important to optimize the SIL decomposition.

5. TABU SEARCH-BASED DESIGN OPTIMIZATION

The problem of scheduling tasks on multiprocessors is known to be NP-complete [Ullman 1975], while the problem of mapping tasks onto a multiprocessor system is proved to be NP-hard [Baruah 2004b]. In order to solve the problem presented in the previous section, we will use the "Mixed-Criticality Design Optimization" (MCDO) strategy from Fig. 7, which is based on a Tabu Search metaheuristic. MCDO takes as input a set of applications Γ (including the SIL information and development costs DC), the SIL decomposition library $\mathcal{L}$ and the set of processing elements $\mathcal{N}$, and returns the implementation Ψ consisting of the SIL decomposition D, the mapping M of tasks to PEs, the set of partitions slices $\mathcal{P}$ on each PE, the assignment ϕ of tasks to partitions and the schedules $\mathcal{S}$ for the applications. Our strategy has 3 steps:

(1) In the first step, we consider that tasks are not decomposed (denoted by D°) and we determine an initial task mapping M°, an initial set of partition slices $\mathcal{P}^\circ$ and an initial assignment of tasks to partitions ϕ°, line 1 in Fig. 7. The initial mapping M° is done in two steps: in the first step, a Greedy algorithm performs the mapping such that the utilization of processors is balanced and minimized. In the second step, the algorithm identifies for each application *orphan* tasks, i.e., tasks mapped to other PEs than any of their immediate successors or predecessors. Next the algorithm remaps these tasks to the same PE as the predecessor or successor that has the largest communication costs, thus minimizing the communication, with the constraint that the utilization of the processors cannot vary more than 25% from the average utilization. $\mathcal{P}^\circ$ consists of a simple straightforward partitioning scheme which allocates for each application $\mathcal{A}_j$ a total time on PE N_i proportional to the utilization of the tasks of $\mathcal{A}_j$ mapped to N_i. The initial assignment ϕ° of tasks to partitions consists of a separate partition for each SIL level in each application, and does not allow partition sharing.

(2) In the second step, we use a Tabu Search meta-heuristic (see Section 5.1) to determine the SIL decomposition D, the task mapping M, the set of partition slices $\mathcal{P}$ and the assignment of tasks ϕ to partitions, such that the applications are schedulable and the development costs are minimized.

(3) Finally, given the SIL decomposition D, the task mapping M, the optimized partitions $\mathcal{P}$ and the assignment ϕ of tasks to partitions obtained in line 2 in Fig. 7, we use a List Scheduling heuristic (see Section 5.2) to determine the schedule tables $\mathcal{S}$ for the applications.

MCDO(Γ, $\mathcal{N}$, $\mathcal{L}$)

1 $< D^\circ, M^\circ, \mathcal{P}^\circ, \phi^\circ >$ = InitialSolution(Γ, $\mathcal{N}$)
2 $< D, M, \mathcal{P}, \phi >$ = TabuSearch(Γ, $\mathcal{N}$, $\mathcal{L}$, D°, M°, $\mathcal{P}^\circ$, ϕ°)
3 $\mathcal{S}$ = ListScheduling(Γ, $\mathcal{N}$, D, M, $\mathcal{P}$, ϕ)
4 **return** $\Psi =< D, M, \mathcal{P}, \phi, \mathcal{S} >$

Fig. 7: Mixed-Criticality Design Optimization strategy

5.1. Tabu Search

TabuSearch(Γ, $\mathcal{N}$, $\mathcal{L}$, D°, M°, $\mathcal{P}^\circ$, ϕ°)

1 *Best* $\leftarrow$ *Current* $\leftarrow < D^\circ, M^\circ, \mathcal{P}^\circ, \phi^\circ >$
2 $L \leftarrow \{\}$
3 **while** termination condition not reached **do**
4 remove tabu with the oldest tenure from L if Size(L) = l
5 // generate a subset of neighbors of the current solution
6 $\mathcal{C} \leftarrow$ GenerateCandidateList(*Current*, Γ, $\mathcal{N}$)
7 *Next* $\leftarrow$ solution from $\mathcal{C}$ that minimizes the cost function
8 **if** Cost(*Next*) $<$ Cost(*Best*) **then**
9 // accept *Next* as *Current* solution if better than the best-so-far *Best*
10 *Best* $\leftarrow$ *Current* $\leftarrow$ *Next*
11 add *Next* to L
12 reset diversification counter, restart counter
13 **else if** Cost(*Next*) $<$ Cost(*Current*) and *Next* $\notin L$ **then**
14 // also accept *Next* as *Current* solution if better than *Current* and not tabu
15 *Current* $\leftarrow$ *Next*
16 add *Next* to L
17 reset diversification counter, restart counter
18 **else**
19 increment diversification counter
20 **end if**
21 **if** diversification counter reached **then**
22 *Current* $\leftarrow$ Diversify(*Current*)
23 empty L
24 increment restart counter
25 **end if**
26 **if** restart counter reached **then**
27 *Current* $\leftarrow$ *Best*
28 empty L
29 **end if**
30 **end while**
31 **return** *Best*

Fig. 8: The Tabu Search algorithm

Tabu Search (TS) [Glover and Laguna 1997] is a meta-heuristic optimization, which searches for that solution which minimizes the *cost function* (see Section 5.1.1 for our cost function definition). Tabu Search takes as input the set of applications Γ, the set of PEs $\mathcal{N}$, the decomposition library $\mathcal{L}$ and the initial solution, consisting of D°, M°, $\mathcal{P}^\circ$, and ϕ°, and returns at the output the best solution found during the design space exploration, in terms of the cost function.

Tabu Search explores the design space by using design transformations (or "moves") applied to the current solution in order to generate neighboring solutions. To escape local minima, TS incorporates an adaptive memory (called "tabu list" or "tabu history"), to prevent the search from revisiting previous solutions, thus avoiding cycling. The size of the tabu list, that is, the number of solutions marked as tabu, is called *tabu tenure*. In case there is no improvement in finding a better solution for a number of iterations, TS uses *diversification*, i.e., visiting previously unexplored regions of the search space. In case the search diversification is unsuccessful, TS will *restart* the search from the best known solution.

Fig. 8 presents the Tabu Search algorithm. Line 1 initializes the *Current* and *Best* solutions to the initial solution formed by the tuple $< D^\circ, M^\circ, \mathcal{P}^\circ, \phi^\circ >$. Line 2 initializes the tabu list L to an empty list. The size l of L, i.e., its *tenure*, is set by the user. The Tabu Search algorithm runs until the termination condition is not reached (see line 3). This termination condition can be, for example, a certain number of iterations or a number of iterations without improvement, considering the cost function [Gendreau 2002]. Our implementation stops the search after a predetermined amount of time, set by the user. In case the tabu list L is filled, we remove the oldest tabu from this list (see line 4).

Since it is infeasible to evaluate all the neighboring solutions (see the discussion in Section 5.1.3), we generate a subset of neighbors of the *Current* solution (line 6), called *Candidate List* and we choose from this *Candidate List*, as the possible *Next* solution, the one that minimizes the cost function (line 7). We accept a solution as the *Current* solution from which the exploration continues if: (1) if it has a cost which is better than the best-so-far solution *Best*, lines 8–12 in Fig. 8, or (2) if it has a better cost then the *Current* solution and it is not "tabu", lines 13–18. If we accept a solution at the *Current* solution, the algorithm resets the diversification and restart counters (lines 12, 17). Otherwise, the algorithm increments the diversification counter. The *Best* and *Current* solutions are updated accordingly, lines 10 and 15, respectively, and the *Next* solution is added to the tabu list L, lines 12 and 16. Note that in the first case we can also accept tabu solutions, which is referred to as "aspiration". In this situation, the already tabu solution in L will be moved to the tail of the list, thus setting its tenure to the size l of the list.

In case the algorithm does not manage to improve the current solution after a given number of iterations, it proceeds to a diversification stage (lines 17–20). During this stage, we attempt to drive the search towards an unexplored region of the design space. Thus, in the Diversify function call, we randomly decompose tasks that have decomposition options specified in the library $\mathcal{L}$, and we randomly re-assign a task from each application, while keeping the same partition tables. The algorithm increments the restart counter after each diversification stage (line 24). If after a preset number of diversification stages, the algorithm is still unable to improve the solution, we restart the search from the best known solution so far (lines 21–24). After a diversification or restart occurs, the tabu list L is emptied.

5.1.1. Cost Function. For each alternative solution visited by TS we use the List Scheduling-based heuristic from Section 5.2 to produce the schedule tables $\mathcal{S}$. We define the response time R_i of an application $\mathcal{A}_i$ as the time difference between the finishing time of the sink node and the start time of the application. $DC(\Gamma)$ is the development cost of the set Γ of all applications (see Section 2.2). We define the cost function of an implementation ψ as:

$$Cost(\psi) = \begin{cases} c_1 = \sum_{\mathcal{A}_i \in \Gamma} \max(0, R_i - D_i) & if\ c_1 > 0 \\ c_2 = DC(\Gamma) & if\ c_1 = 0 \end{cases} \quad (1)$$

If at least one application is not schedulable, there exists one R_i greater than the deadline D_i, and therefore the term c_1 will be positive. However if all the applications are schedulable, this means that each R_i is smaller than D_i, and the term $c_1 = 0$. In this case, we use c_2 as the cost function, since when the applications are schedulable, we are interested to minimize the development cost.

5.1.2. Design Transformations. As previously mentioned, the exploration of the design space is done by applying design transformations (moves) to the current solution *Current*. We use one *re-*

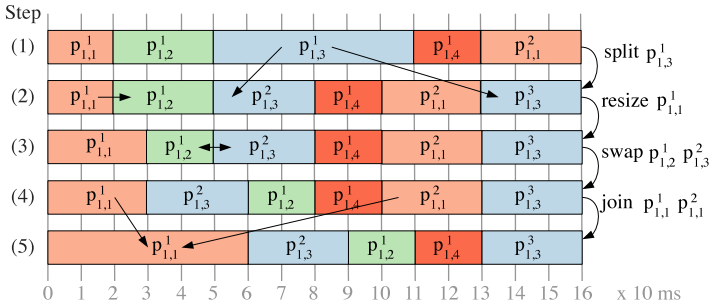


Fig. 9: Partition slice move examples

assignment move, which changes the assignment of a task to another partition and four types of moves applied to partition slices: *resize*, *swap*, *join* and *split*. We also employ SIL decomposition moves, namely *decompose* and *recompose*.

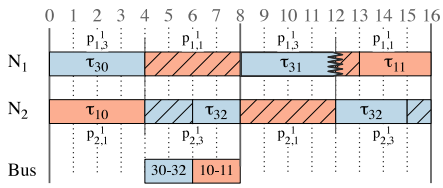
Let us first discuss the moves applied to partition slices. The *resize* move, as its name implies, resizes the selected partition slice. This is done either by increasing the size of the partition slice at the expense of a neighboring partition slice, or by decreasing it and giving the extra space to a neighboring slice. The amount with which the slice can be resized is randomly chosen, but we have imposed an upper limit (half the size of the partition slice). The *swap* move swaps the chosen partition slice with another randomly chosen partition slice. The *join* move joins two partition slices belonging to the same application, while the *split* move splits a partition slice into two, and adds the second slice to the end of the MF.

Fig. 9 depicts the basic slice moves as they are sequentially performed on a single PE, namely N_1. As mentioned, the notation $p^k_{i,j}$ means the k^{th} partition slice of the application $\mathcal{A}_j$ on the processing element N_i. There are 4 applications, numbered from 1 to 4, and the first application has 2 partition slices, $p^1_{1,1}$ and $p^2_{1,1}$. The current partitioning solution is presented in Step 1 in Fig. 9. The first move is the *split* move, which is performed on the partition slice $p^1_{1,3}$ belonging to $\mathcal{A}_3$. The slice is split in two equal parts, and the resulting slice is added to the end of the MF. The second move is a *resize* with 10 ms, which affects $p^1_{1,1}$ at the expense of $p^1_{1,2}$. The third move is a *swap* of slices $p^1_{1,2}$ and $p^2_{1,3}$. The result is shown in the 4^{th} step. The last move is a *join* move and as previously mentioned, it can be applied only to partition slices belonging to the same application. For this move we chose the $p^1_{1,1}$ and $p^2_{1,1}$ slices.

The *re-assignment* move re-assigns a task to another partition. The partition can be an existing one, or newly created. The partition may be on another PE, thus, implicitly, the re-assignment move will also re-map the task. The re-assignment move does not prevent partition sharing by tasks of different SILs. In case the move will lead to sharing, we elevate tasks as required, and update the development costs accordingly. If a re-assignment move results in an empty partition, the partition is deleted and its assigned CPU time is distributed to a randomly chosen partition. As a result, the algorithm creates and deletes partitions and partition slices, as well as resizes them, on the fly as needed, depending on the task re-assignment moves.

The SIL decomposition moves are applied to the tasks which have decomposition alternatives specified in the library $\mathcal{L}$. The *decompose* move selects a random decomposition option from the library. The *recompose* move is applied to a task τ_i, and it reverts the task to its initially proposed model, thus undoing any *decompose* moves that may have affected τ_i. These moves are applied during the diversification phase (line 18 in Fig. 8) to randomly selected tasks.

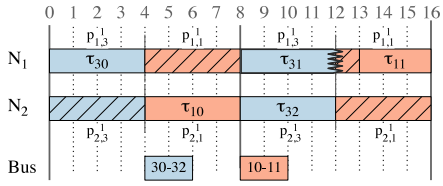
Our algorithm relies on a *tabu list* with tabu-active attributes, that is, it does not remember complete solutions in the list L, but rather attributes of the moves that generated the tabu solutions. In case of the resize and the swap moves, tabu-active attributes are the involved partition slices. For



Tenure	Tabu
5	*resize* $p_{2,1}^1$ and $p_{2,3}^1$ on N_2
4	*join* $p_{1,1}^1$ and $p_{1,1}^2$ from $\mathcal{P}_{1,1}$ on N_1
3	*resize* $p_{1,1}^1$ and $p_{1,3}^1$ on N_1
2	*swap* $p_{1,1}^1$ and $p_{1,1}^2$ on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 7

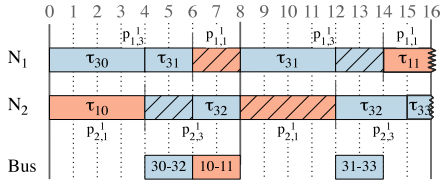
(a) Current solution



Tenure	Tabu
5	*resize* $p_{2,1}^1$ and $p_{2,3}^1$ on N_2
4	*join* $p_{1,1}^1$ and $p_{1,1}^2$ from $\mathcal{P}_{1,1}$ on N_1
3	*resize* $p_{1,1}^1$ and $p_{1,3}^1$ on N_1
2	*swap* $p_{1,1}^1$ and $p_{1,1}^2$ on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 7

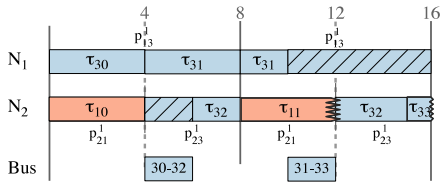
(b) Swap the partitions on N_2, results in a solution which is not better than the current solution



Tenure	Tabu
5	*resize* $p_{1,1}^1$ and $p_{1,3}^1$ on N_1
4	*resize* $p_{2,1}^1$ and $p_{2,3}^1$ on N_2
3	*join* $p_{1,1}^1$ and $p_{1,1}^2$ from $\mathcal{P}_{1,1}$ on N_1
2	*swap* $p_{1,1}^1$ and $p_{1,1}^2$ on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 4

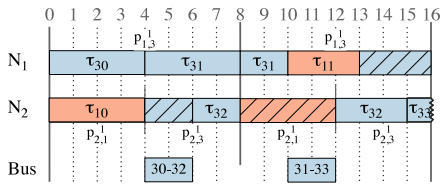
(c) Resize τ_{31}'s partition. Best solution so far, although it is a tabu move. The tabu status is "aspired"



Tenure	Tabu
5	*resize* p_{21}^1 and p_{23}^1 on N_2
4	*join* p_{11}^1 and p_{11}^2 from $\mathcal{P}_{1,1}$ on N_1
3	*resize* p_{11}^1 and p_{13}^1 on N_1
2	*swap* p_{11}^1 and p_{11}^2 on N_1
1	*re-assign* τ_{11} from $\mathcal{P}_{2,1}$ to $\mathcal{P}_{1,1}$

Cost = 4

(d) Re-assign τ_{11} to N_2. Tabu move and does not improve the solution



Tenure	Tabu
5	*re-assign* τ_{11} *from* $\mathcal{P}_{1,1}$ *to* $\mathcal{P}_{1,3}$
4	*resize* $p_{2,1}^1$ *and* $p_{2,3}^1$ *on* N_2
3	*join* $p_{1,1}^1$ *and* $p_{1,1}^2$ *from* $\mathcal{P}_{1,1}$ *on* N_1
2	*resize* $p_{1,1}^1$ and $p_{1,3}^1$ on N_1
1	*swap* $p_{1,1}^1$ and $p_{1,1}^2$ on N_1

Cost = 3

(e) Re-assign τ_{11} to $\mathcal{A}_3$'s partition on N_1. Best solution so far

Fig. 10: Moves and tabu history

the split and join moves, the tabu-active attribute is the partition the move was performed on. As for the re-assignment move, the attributes are the re-assigned task and the involved partitions.

Let us illustrate in Fig. 10 how Tabu Search works. We consider applications $\mathcal{A}_1$, with tasks τ_{10} and τ_{11}, and $\mathcal{A}_3$, with tasks τ_{30}–τ_{33}, from Fig. 1, with their periods and deadlines equal to 16. The size of the major frame T_{MF} is set to 8 and T_{cycle} is 16. We are interested in implementing these applications on an architecture with two PEs, N_1 and N_2. Let us assume that we are running our TS and the current solution, which is also the best-so-far solution, is the one presented in in Fig. 10a.

The mapping and assignment of tasks in this solution is as follows. $\tau_{10} \in \mathcal{A}_1$ is assigned to partition $\mathcal{P}_{2,1}$ on N_2 (composed of partition slice $p^1_{2,1}$), while τ_{11} is assigned to partition $\mathcal{P}_{1,1}$ on N_1 (with partition slice $p^1_{1,1}$). In the case of application $\mathcal{A}_3$, τ_{30} and τ_{31} are assigned to partition $\mathcal{P}_{1,3}$ on N_1 (of slice $p^1_{1,3}$), while τ_{32} and τ_{33} are assigned to $\mathcal{P}_{2,3}$ on N_2 (with slice $p^1_{2,3}$). Note that this solution is not schedulable, since tasks τ_{31} and τ_{33} from $\mathcal{A}_3$ do not fit into the schedule. Each of the figures from Fig. 10b–10e presents a neighboring solution generated from the current solution in Fig. 10a, and are intended to illustrate moves performed by TS and how the tabu list is updated. None of these solutions are schedulable, but we can see improvements in the cost function, which will drive the search to a schedulable solution.

Next to each solution we present the value of the cost function associated to the solution. Since none of these solution are schedulable, the value of the cost function is the term c_1 from Eq. 1. Furthermore, we also present for each solution the updated tabu list (referred to as L in Fig. 8). Fig. 10a presents the current tabu list. Fig. 10b–10e present the updated list, that will be used in case the associated solution is chosen as the as the *Next* solution (see Fig. 8). For this example, the tabu tenure l is 5. The tabu most recently added to the list has the highest tenure, while the oldest tabu in the list has the lowest tenure. For example, in Fig. 10a, the most recently added tabu to the list has a tenure of 5. This tabu is associated to the move that generated this solution, namely a *resize* move, and the involved partition slices are $p^1_{2,1}$ and $p^1_{2,3}$.

Fig. 10b presents a neighboring solution obtained from Fig. 10a by swapping on N_2 the partition slices $p^1_{2,1}$ and $p^1_{2,3}$ assigned to $\mathcal{A}_1$ and $\mathcal{A}_3$, respectively. This move does not improve the solution, i.e., the value of the cost function is 7 in both cases, thus, the move is ignored and the tabu list is not updated. Fig. 10c shows the solution obtained from Fig. 10a obtained by resizing the partition slice $p^1_{1,3}$ on N_1. In this solution, $p^1_{1,3}$ is increased, while the size of $p^1_{1,1}$ is decreased. This solution was generated by a move that is tabu. Because this solution is better than the best-so-far solution shown in Fig. 10a, in terms of the cost function (the value of the cost function is 4 in the new solution, compared to 7 in Fig. 10a) the tabu status of the move is ignored or "aspired" (TS can choose tabu moves, if the resulting solutions are better than the *Best* known solution). The updated tabu list, in case the search will continue with this solution as the *Current* solution, is presented next to the solution.

The solution in Fig. 10d is obtained by re-assigning task τ_{11} from partition $\mathcal{P}_{1,1}$ on N_1 (composed of partition slice $p^1_{1,1}$) in Fig. 10a to $\mathcal{P}_{2,1}$ on N_2 (of slice $p^1_{2,1}$). After re-assigning τ_{11}, partition $\mathcal{P}_{1,1}$ on N_1, composed of $p^1_{1,1}$, has no tasks assigned to it, therefore it is deleted and the algorithm "transfers" the CPU time of $\mathcal{P}_{1,1}$ to $\mathcal{P}_{1,3}$ (composed of $p^1_{1,3}$). Thus, on N_1, there is only one partition slice executing, i.e., $p^1_{1,3}$. Although this move does improve the solution presented in Fig. 10a in terms of the cost function, it is not better than the solution from Fig. 10c, and hence, it is ignored. Fig. 10e presents a solution obtained by re-assigning τ_{11} from partition $\mathcal{P}_{1,1}$ on N_1 in Fig. 10a to $\mathcal{P}_{1,3}$. Similar to the solution from Fig. 10d, since partition $\mathcal{P}_{1,1}$ has no tasks assigned to it, it is deleted and its CPU time given to $\mathcal{P}_{1,3}$. Furthermore, τ_{11} shares the partition with τ_{30} and τ_{31}. Since τ_{11} is a task with SIL 3, and tasks τ_{30} and τ_{31} are SIL 2 tasks, the two tasks from $\mathcal{A}_3$ have to be elevated to SIL 3, thus increasing the development costs of the system. This move does not result in a schedulable solution, but it improves the solution in terms of cost function. The value of the cost function in this case is 3, and is better than the other neighboring solutions. The search will continue with this solution as the *Current* solution, and the tabu list will be accordingly updated.

5.1.3. Candidate List. The neighborhood of the *Current* solution is composed of all the solutions which are "one move away", that is, obtained by applying a move to the *Current* solution. To decide which move to select as the *Next* solution, we need to determine which of the neighbors minimizes the cost function (line 7 in Fig. 8). Calculating the cost function (Eq. 1) means determining the schedule tables for all the applications (term c_1 in Eq. 1) and, if they are schedulable, a sumation of the development costs for all tasks (term c_2). Since the size of a neighborhood is large, calculating

the cost function for every neighbor would take a very long time, rendering the search process infeasible. Instead, only a part of the neighborhood is considered, and neighbors are placed on a so called *candidate list* $\mathcal{C}$. One option is to select randomly the neighbors to be placed on the candidate list. However, we use a heuristic approach that selects those neighbors which have a higher chance to lead quickly to good quality solutions.

GenerateCandidateList(*Current*$,\Gamma,\mathcal{N}$**)**

1 $\mathcal{C} \leftarrow \{\}$
2 **for** $PE_i \in \mathcal{N}$ **do**
3 **for all** move $\in$ {resize, swap, join, split} **do**
4 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ apply *move* to a random partition slice on PE_j in *Current*$\}$
5 **end for**
6 $\mathcal{C} \leftarrow \mathcal{C} \cup \{ New | New \leftarrow$ *resize* most oversized partition on PE_j in *Current*$\}$
7 $\mathcal{C} \leftarrow \mathcal{C} \cup \{ New | New \leftarrow$ *resize* most undersized partition on PE_j in *Current*$\}$
8 **end for**
9 **if** perform moves on tasks **then**
10 **for all** applications $\mathcal{A}_i$ that missed their deadlines **do**
11 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task $\tau_j \in \mathcal{A}_i$ to random partition in *Current*$\}$
12 **end for**
13 **for all** $PE_i \in \mathcal{N}$ **do**
14 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task τ_j from PE_i to a random partition on $PE_k \neq PE_i$ in *Current*$\}$
15 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task τ_j from PE_i to another partition on PE_i in *Current*$\}$
16 $\mathcal{C} \leftarrow \mathcal{C} \cup \{New | New \leftarrow$ *re-assign* random task τ_j from PE_i to another application's partition in *Current*$\}$
17 **end for**
18 **end if**
19 **return** $\mathcal{C}$

Fig. 11: Algorithm to generate the candidate list $\mathcal{C}$

The candidate list generation algorithm (GCL) is presented in Fig. 11. GCL takes as input the *Current* solution, the set of applications Γ and the set of processing elements $\mathcal{N}$, and returns a list $\mathcal{C}$ of candidate solutions. GCL is called on line 6 in our Tabu Search from Fig. 8.

GCL starts with an empty candidate list $\mathcal{C}$ (line 1 in Fig. 11). The neighbors placed in $\mathcal{C}$ are obtained by performing moves on the *Current* solution. The following moves are considered. On each PE, GCL performs partition related moves (resize, swap, join and split moves) on random partition slices (lines 3–5). On each PE, GCL chooses as a candidate the most oversized partition, that is, the partition with the lowest ratio of used CPU time compared to actual allocated time (line 6), and resizes (shrinks) this partition. This is done to transfer "unused" CPU time from an oversized partition to another partition, in the hope of improving the overall schedulability of the system. Similarly, there might exist situations where we have undersized partitions, that is, partitions that have more tasks assigned than allocated CPU time, or partitions where the allocated time is not enough for all the assigned tasks to execute before their deadline. For such situations, on each PE, GCL selects the most undersized partition, i.e., the partition with the highest ratio of required CPU time compared to the actual allocated time, and resizes this partition, increasing its size (line 7). Such an example is given in Fig. 10c, obtained from Fig. 10a. The most undersized partition in Fig 10a, on PE N_1 is partition $\mathcal{P}_{1,3}$ containing partition slice $p^1_{1,3}$. This partition has an allocated time of 8 time units during the MF, and has assigned to it tasks τ_{30} and τ_{31}, requiring 10 time units for execution. Thus, it has a ratio of required to allocated CPU time of 125%. The other partition

on N_1, $\mathcal{P}_{1,1}$, has only τ_{11} assigned to it, which requires only 3 time units for execution, out of the 8 allocated to the partition. The ratio of required to allocated CPU time for this partition is only 37.5%. Hence, the most undersized partition, i.e., $\mathcal{P}_{1,3}$, is increased, by transferring CPU time from partition slice $p^1_{1,1}$ to $p^1_{1,3}$. The candidate solution generated by this move is presented in Fig. 10c, and is better than the solution shown in Fig. 10a, in terms of the cost function.

The diversification stage presented in Fig. 8, line 18, randomly re-assigns a task from each application, while keeping the same partition tables. Furthermore, during this phase, randomly selected tasks that have decomposition options specified in the decomposition library $\mathcal{L}$, are decomposed. The introduction of the decomposed tasks may decrease the schedulability of the diversified solution. In order to allow TS to improve on the schedulability by adapting the partition table to the new mapping scheme and to the decomposed tasks, we do not allow any re-assignment moves for a certain number of iterations. This condition is shown in line 9, in Fig. 11. Thus, we force the TS to explore this new design space area, while keeping the assignment of tasks to partitions fixed. When this restriction is lifted, GCL focuses on the applications that miss their deadlines, in order to make them schedulable (lines 10–12 in Fig. 11). For this, GCL selects a random task from each application missing its deadline, and re-assigns it to another partition. Furthermore, on each PE we perform three types of re-assign moves (lines 13–17), in hope to thoroughly explore the design space. GCL re-assigns a random task τ_i to another PE, but to the same application's partition (line 14); a random task to the same PE, but to another partition (line 15); and another task to another PE, to another application's partition (line 16).

5.2. List Scheduling

The applications are scheduled using static-cycling non-preemptive scheduling. We use a List Scheduling (LS)-based heuristic [Sinnen 2006] to determine the schedule tables $\mathcal{S}$ for each application. Adam et al. [1974] have shown that critical path-based List Scheduling heuristics result in schedules that are "within 5 percent of the optimal execution time in 90 percent of the cases". LS heuristics use a sorted priority list, L_{ready}, containing the tasks ready to be scheduled. A task τ_i is *ready* if all the predecessor tasks have finished executing and all the incoming messages are received. We use the Modified Partial Critical Path priority function [Pop et al. 2004] to sort L_{ready}.

We have modified the classical LS algorithm to take into account the partitions. Thus, when scheduling a task, we are allowed to use only the corresponding partitions slices from $\mathcal{P}$. Our LS is implemented as a loop. During each iteration, the LS algorithm checks which partition is active during the current step. We use a priority list $L^{\mathcal{P}_{i,j}}_{ready}$ for each partition P_j on processor N_i. LS selects the task τ_n with the highest priority from the $L^{\mathcal{P}_{i,j}}_{ready}$ and schedules it in the current partition slice $p^k_{i,j}$. If a partition slice finishes before the task task τ_n has completed its execution (as is the case with $\tau_{31} \in \mathcal{A}_3$ in Fig. 3b), τ_n is suspended at the end of the slice, $C^{N_i}_n$ is updated with the remaining time to execute, and τ_n is not removed from the ready list. Then LS continues scheduling tasks from the ready list of the next partition. When the processor is assigned to the next partition slice $p^{k+1}_{i,j}$ of P_j, LS will continue with scheduling task τ_n. The partition switch overhead t_O is taken into account at each partition switch by reducing the actual duration of each partition slice with t_O. LS also derives the schedules tables for the messages on the bus.

6. EXPERIMENTAL EVALUATION

For the evaluation of our proposed "Mixed-Criticality Design Optimization" (MCDO) algorithm we used 7 synthetic benchmarks and 2 real-life case studies. The MCDO algorithm was implemented in Java (JDK 1.6), running on SunFire v440 computers with UltraSPARC IIIi CPUs at 1.062 GHz and 8 GB of RAM.

In the first set of experiments we were interested to evaluate the proposed MCDO in terms of its ability to find schedulable implementations. Thus, we have used 3 synthetic benchmarks with 3 to 5 mixed-criticality applications (with a total of 15 to 41 tasks). We have used MCDO to implement

Table II: Comparison of MO+PO, MPO and MCDO (run time: 480 minutes)

Set	Benchmark	Apps	Tasks	PE	MO+PO	MPO		MCDO	
					Sched. Apps	Sched. Apps	δ_{Sched} (%)	Sched. Apps	δ_{DC} (kEuro)
1	1	3	15	2	2	2	450	All	59
	2	4	34	4	0	3	3600	All	19
	3	5	41	5	3	All	235	–	–
2	4.1	3	20	4	All	All	1.10	–	–
	4.2	4	30	4	All	All	23.96	–	–
	4.3	5	34	4	4	All	13.27	–	–
	4.4	6	39	4	3	5	208.11	All	470
3	5.1	20	210	15	16	All	1912	–	–
	5.2	25	265	15	24	All	27647	–	–
	5.3	30	333	22	18	28	96.51	All	158
	5.4	35	371	20	33	All	1602	–	–
	5.5	40	270	25	36	38	42.29	All	89
3	consumer	2	12	3	0	1	343.45	All	123
	networking	4	13	3	2	2	31.78	All	40

these applications on architectures with 2 to 5 processing elements. The execution times and message lengths were assigned randomly within the 1 to 19 ms and 1 to 5 bytes ranges, respectively. The details of each benchmark, namely the benchmark number, the number of applications, the number of tasks and the number of processing elements are presented in Table II, columns 2–5. For all the experiments, we used the decompositions in Table I.

We were interested to compare the number of schedulable implementations found by MCDO with two other setups. In one of the setup, SIL decomposition is not used and the sharing of partitions by tasks of different criticality levels is not allowed, but mapping and partitioning optimization is performed simultaneously. Let us call this simultaneous "mapping and partitioning optimization", MPO. In the other setup, sharing and decomposition are not allowed, and in addition, mapping optimization (MO) is performed separately from partitioning optimization (PO). We call such an approach MO+PO.

MO+PO and MPO are based on the MCDO strategy presented in Fig. 7, and use the same Tabu Search for the optimization. The difference is in the types of moves performed by TS: there are only mapping moves for MO (without considering partitions), we use only partition-related moves in PO, considering mapping fixed, as determined by MO. In addition, MPO does not allow decomposition moves and re-assignment moves that would lead to partition sharing by mixed-criticality tasks. Further, MO, PO and MPO use a cost function where we do not consider development costs (the term c_2 in Eq. 1), which are constant in their case since we do not elevate or decompose tasks:

$$Cost(\Psi) = \begin{cases} c_1 = \sum_{\mathcal{A}_i \in \Gamma} \max(0, R_i - D_i) & if\ c_1 > 0 \\ c_2 = \sum_{\mathcal{A}_i \in \Gamma} (R_i - D_i) & if\ c_1 = 0. \end{cases} \quad (2)$$

If at least one application $\mathcal{A}_i$ is not schedulable, there exists one application completion time R_i greater than the deadline D_i, and therefore the term c_1 will be positive. However if all applications are schedulable, this means that each R_i is smaller than D_i, and the term $c_1 = 0$. In this case, we use c_2 as the "degree of schedulability", since it can distinguish between two schedulable solutions. The solution with a higher "degree of schedulability", i.e., a smaller c_2 value, is preferred.

The termination condition of our Tabu Search is a time limit given by the designer. The time imposed for each individual experiment is 480 minutes. To determine an appropriate value, we have run our TS for long periods, e.g., for 2 days and then determined the shortest time limit that can produce a result within 5% of the result obtained in 2 days, in our case 480 minutes. The results for the first set of experiments are presented in Table II in the rows corresponding to "Set 1". The

number of schedulable applications, resulted after implementing the system using MO+PO, MPO and MCDO are reported in columns 6, 7 and 9, respectively, labelled with the respective acronym and "Sched. apps.".

As we can see from the comparison between MO+PO and MPO, there is a significant improvement in the number of schedulable applications if the optimization of mapping is considered at the same time with the optimization of partitioning. For example, for the second benchmark (benchmark 2 in Set 1) with 4 applications mapped to 4 PEs, MO+PO is unable to successfully schedule any of the applications. MPO, which performs mapping and time optimization simultaneously, is able to schedule 3 out 4 applications. We have also compared MPO and MO+PO in terms of the degree of schedulability, i.e., the cost function captured by Eq. 2. The percentage improvement δ_{sched} of MPO over MO+PO is presented in column 8 and is computed as

$$\delta_{sched} = \frac{Cost(MPO) - Cost(MO+PO)}{Cost(MO+PO)} \times 100 \tag{3}$$

where Cost(MPO) and Cost(MO+PO) are the values of the cost function for the solutions obtained using the MPO and MO+PO strategies, respectively. An improvement in the degree of schedulability means that there is more slack available in the schedule, which can be used for future upgrades, for example.

If MPO produces a schedulable solution, i.e., the applications are schedulable without SIL decomposition or partition sharing, we do not have to run MCDO. This is indicated in the table using a dash "–" in the MCDO columns. However, MPO is not able to find schedulable implementations in the first two cases. In such situations, MCDO, which optimizes the SIL decompositions and the partition sharing at the same time with mapping and partitioning, can find schedulable implementations in all cases.

Once a schedulable implementation is found by using decomposition and elevation, the cost function from Eq. 1 will drive MCDO to solutions that minimize the development cost. Elevation for partition sharing will increase the development costs, whereas SIL decomposition has the potential to reduce these costs. The increase δ_{DC} in development cost that we have to pay in order to find schedulable implementations, compared to MPO which does not perform SIL elevation or decomposition, is reported in the last column of Table II.

In the second set of experiments, labeled "Set 2" in Table II, we were interested to see how MCDO performs compared to MO+PO and MPO as the utilization of the system increases. Thus, we have an increasing number of mixed-criticality applications, from 3 to 6, on the same architecture of 4 PEs. The average utilization of the system, per processor, increases from 46.7% in benchmark 4.1 to 77.9% in benchmark 4.4. As we can see, for the smaller benchmarks of 3 and 4 applications (benchmarks 4.1 and 4.2, respectively), MO+PO is able to find schedulable implementations. Optimizing the mapping and time partitions using MPO leads to more schedulable implementations, i.e., "All" applications are schedulable in benchmark 4.3. However, as the system utilization increases, as is the case for the largest benchmark in this set (4.4), where we used 6 applications on 4 PEs, only MCDO, which considers decomposition and elevation to allow partition sharing by tasks of mixed-criticality, is able to find schedulable solutions. Therefore, MCDO is able to integrate successfully more mixed-criticality applications on the same integrated architecture, thus saving product unit costs by avoiding costly architecture upgrades across the product line.

In the third set of experiments "Set 3", we wanted to see how the optimization strategies perform on large test cases. In the new test cases, the number of applications vary between 20 to 40, with 210 to 371 tasks, and are mapped on 15 to 25 processing elements. The results confirm the findings of the previous sets. Moreover, we see that our optimization handle well also large test cases.

We were interested to determine how close are the results obtained by MCDO to the optimal result. We have run an exhaustive search for test case 1 in Table II and obtained thus the optimal solution. Running MCDO for 480 minutes on this benchmark, we have been able to obtain a result which has the same cost function value as the optimal solution.

Finally, we have also used 2 real life benchmarks derived from the Embedded Systems Synthesis Benchmarks Suite [Dick 2005] version 0.9. We have used the *consumer-cords* and *networking-cords* benchmarks. In both cases we were interested in implementing the applications to an architecture of 3 PEs. The results obtained from these real-life benchmarks are reported in the last 2 lines in Table II and confirm the results of the synthetic benchmarks.

7. RELATED WORK

There is a large amount of research on hard real-time systems [Kopetz 2011b; Buttazzo 1997], including task mapping to heterogeneous architectures [Braun et al. 2001]. Researchers have addressed systems with mixed *time*-criticality requirements, showing how Time Triggered (TT)/Event Triggered (ET) tasks or hard/soft real-time tasks can be integrated onto the same platform.

In the context of mixed TT/ET systems, Pop et al. [2008a] have shown how the static schedules can be optimized such that both the TT applications (scheduled using SCS) and the ET applications (scheduled using FPS) are schedulable. Their approach could be extended to constrain the TT schedules to follow a given partitioning. They have later addressed the problem of mapping and partitioning, but in their context partitioning means deciding which tasks should be TT and which ET [Pop et al. 2006]. While in [Pop et al. 2006, 2008a] TT and ET tasks share the same processor, the work in [Pop et al. 2004] considers that TT and ET tasks are implemented on different *clusters*. In this context, partitioning means deciding in which cluster (TT or ET) to place a task.

Researchers have shown how to integrate mixed hard/soft real-time tasks onto the same platform. The order of tasks is decided by quasi-static scheduling in [Cortés et al. 2004] (several schedules are determined offline, and are activated online depending on when tasks finish executing), such that the hard tasks meet their deadlines and the total "utility" of soft tasks is maximized. This work has been extended by Izosimov et al. [2008] to handle transient faults, by switching online to backup recovery schedules. Soft real-time tasks can be integrated in fixed-priority preemptive scheduling using the Constant Bandwidth Server (CBS) [Abeni and Buttazzo 1998], where the server is a hard task providing a desired level of service to soft tasks. Thus, the CBS-servers provide a time-partitioning between hard and soft tasks. The optimization of CBS-server capacity in the context of mixed hard and soft real-time tasks has been addressed by Saraswat et al. [2010], such that the hard tasks are schedulable and the quality of service for the soft tasks is maximized.

The problem of the optimization of time-partitions has been addressed at the bus level, but without considering partitions at the processor level. Researchers have shown how a Time-Division Multiple Access bus such as the TTP [Pop et al. 1999] and a mixed TT/ET bus such as FlexRay [Pop et al. 2008b] can be optimized to decrease the end-to-end delays. The optimization implies deciding on the sequence and length of the communication slots.

Lee et al. [2000] consider an IMA-based system where all tasks are scheduled using FPS. The time-partition optimization problem is formulated as a static cyclic scheduling problem, where the partitions are statically scheduled such that the FPS tasks are schedulable. A similar approach to IMA is used in the DEOS operating system [Binns 2001], with the difference that FPS is used for scheduling both the partitions (which are normally scheduled using SCS) and the tasks. Binns [2001] has proposed several slack-stealing approaches, where the unused time in one partition is given to the other partitions, thus the partitions are implicitly adjusted online.

There are several works where mixed-criticality tasks are addressed, mostly targeting uniprocessor systems. Vestal [2007] was the first to extend the task model to include criticality-level dependent Worst-Case Execution Times (WCET). Furthermore, he proposes two fixed-priority preemptive scheduling algorithms. Vestal's assumption is that the method used to determine the WCET, and thus the accuracy of the value, depends on the criticality level of the task. For example, if for the highest criticality level a WCET analysis is suggested, for the lowest criticality the WCET value can be obtained using simulations. In our work, we assume that a task has the same WCET value, regardless of the criticality level. Baruah and Vestal [2008] extend the work from [Vestal 2007] and propose for sporadic tasks sets a hybrid-priority scheduling policy [Baruah and Fisher 2008], which includes features of the Earliest Deadline First (EDF) policy as well. Baruah et al. [2010] propose

a task model that can capture mixed-criticality functions, together with an associated schedulability analysis.

Several papers [Li and Baruah 2010; Baruah and Fohler 2011; Baruah et al. 2011b] base their work on the assumption that the Certification Authorities (CAs) consider a more pessimistic WCET for the tasks than the system designer, leading to inefficient usage of computing resources. Thus, for each task they take into account two WCETs: a HI WCET, pessimistic, considered by the CA, and a less pessimistic LO WCET expected by the system designer. The work in [Li and Baruah 2010] proposes an algorithm for on-line priority-based scheduling of mixed-criticality sporadic tasks on uniprocessors. If during a busy period, a high criticality task exhibits HI WCET behavior, that is, its execution time is larger than the LO WCET assumed by the designer, the low criticality tasks are discarded, in order to ensure the necessary CPU time to all the high criticality tasks. Baruah et al. [2011b] introduce a novel scheduling algorithm using Fixed Priority on uniprocessor mixed-criticality systems which takes into account the criticality level of each task, evaluate three possible priority assignment schemes, and propose an associated response time analysis.

Baruah and Fohler [2011] offer a solution using Time-Triggered scheduling to the problem in [Baruah et al. 2010; Li and Baruah 2010]. They propose a "mode change" approach: using the algorithm from [Li and Baruah 2010], they compute offline two schedules, a "certification mode" schedule considering the HI WCET behavior, and a "designer mode" schedule considering the LO WCET behavior. In case a task overruns its LO WCET, a mode change is triggered, and the system runs using the "certification mode" schedule.

In [de Niz et al. 2009], the authors discuss the issue of "criticality inversion", similar to the classical priority inversion problem, and propose a "zero-slack scheduling" scheme for such a context. Mollison et al. [2010] propose a scheduling architecture for mixed-criticality tasks on multicore systems. In this architecture, the high criticality tasks are considered *slack generators*, as the WCET predictions are deemed overly pessimistic, and the authors assume that these tasks will "use only a small fraction of the execution time budgeted for them". Employing slack shifting, this architecture considers the low criticality tasks *slack consumers* and are allowed to execute during the slack if it will not have a negative impact on the timing of the high criticality tasks. Li and Baruah [2012] propose a scheduling algorithm for mixed-criticality sporadic tasks implemented on homogeneous multiprocessor platforms, were task migration is permitted. Their algorithm is based on the EDF-VD [Baruah et al. 2011a] uniprocessor scheduling algorithm and fpEDF [Baruah 2004a] global scheduling algorithm.

As mentioned, a SIL captures the required level of risk reduction, and will dictate the development processes and certification procedures that have to be followed. SILs differ slightly among areas. For example, the avionics area uses five "Design Assurance Levels" (DAL), from DAL E (lest critical) to DAL A (most critical), while ISO 26262 specifies for the automotive area four "Automotive Safety Integrity Levels" (ASIL), from ASIL A (least critical) to ASIL D (most critical). However, the approach presented in this paper is applicable to all safety-critical areas, regardless of the standard. SILs are assigned to tasks, from SIL 4 (most critical) to SIL 0 (non-critical).

Giannopoulou et al. [2013] propose for multicore platforms a mapping optimization and scheduling strategy that takes into account the effects of resource sharing on execution times, by restricting the cores to synchronize and execute tasks of the same criticality levels. In Giannopoulou et al. [2014] they extend the previous work with an optimization method that statically maps task data and communication buffers to the banks of the shared memory, to reduce the effect of bank sharing on the task response times.

Standards provide checklists of objectives required to be fulfilled for each SIL. Depending on the SIL, the standard may also impose that some objectives to be satisfied with independence, to ensure an unbiased evaluation and to avoid misinterpretation of the requirements [RTCA DO-178B 1992]. For example, for the verification process, independence is achieved by using tools and personnel other than those used throughout the development process.

SIL 0 functions are non-critical and do not impact the safety of the systems, thus are not covered by the standards. In the case of SIL 1, the processes are similar to those covered by quality man-

agement standards such as ISO 9001 [ISO 9001 2008]. SIL 2 involves more reviewing and testing. SIL 3 is significantly more difficult, and requires "semi-formal" methods. SIL 4 often mandates formal methods, increasing further the development costs.

The assessment of conformity to the checklist of objectives has to be performed by independent assessors. For SIL 1 is enough to have an independent person, whereas for SIL 2 an independent department is required. In the case of SIL 3 and SIL 4, an independent organization has to be used. Moreover, the number of objectives that have to be satisfied with independence is also growing. For example, in the case of DO-178B, the main difference between DAL A and DAL B is the number of objectives to be satisfied with independence: 25 out of 66 objectives are required for DAL A to be satisfied with independence, while for DAL B it is only 14 out of 66.

In this paper we have also addressed the issue of SIL decomposition. SIL allocation is typically a manual process, which is done after performing hazard and risk analysis [Storey 1996], although a few researchers have proposed automatic approaches for SIL allocation [Papadopoulos et al. 2010]. Parker et al. [2013] and Azevedo et al. [2013] have proposed Genetic Algorithm and Tabu Search-based metaheuristics for SIL decomposition. These algorithms aim to reduce the development costs and focus on deriving fault-tolerant architectures. The safety of the resulted architecture is evaluated using Fault-Tree Analysis.

Our work has been motivated by the need to reduce the certification costs, which add a 25 to 100% overhead to the development costs [IBM 2010]. Hence, we were interested in realistic assumptions that are accepted by the current certification practice. These assumptions have been validated in the RECOMP project ("Reduced Certification Costs Using Trusted Multi-core Platforms") [Pop et al. 2013], where certification authorities such as TÜV SÜD were participants.

8. CONCLUSIONS

In this paper we have presented a Tabu Search-based approach for the optimization of mixed-criticality applications on cost-constrained partitioned architectures. The architectures consist of a set of heterogeneous processing elements interconnected by a broadcast bus. With partitioning, tasks of different criticality are allowed to use the PEs only during predetermined time slots, and are thus separated in both space and time. We have considered that tasks and messages are scheduled using Static Cyclic Scheduling.

We were interested to derive schedulable implementations that minimize the development costs. We have seen that significant improvements can be gained considering the optimization of task mapping to PEs at the same time with the optimization of partitions, which decides the sequence and size of the time partition time slots on each PE.

However, there are situations when finding schedulable implementations on cost-constrained architectures is only possible if we allow tasks of different criticality to share a partition. This implies the elevation of tasks to the highest Safety-Integrity Level of a partition, which leads to increased development costs. This increase in development costs can be partially avoided by using SIL decomposition. Our optimization approach is able to find schedulable implementations on cost-constrained architecture, which minimizes the development costs.

REFERENCES

L. Abeni and G. Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of Real-Time Systems Symposium*. 4 –13.

Thomas L. Adam, K. M. Chandy, and J. R. Dickson. 1974. A Comparison of List Schedules for Parallel Processing Systems. *Comm. ACM* 17, 12 (Dec. 1974), 685–690. DOI:http://dx.doi.org/10.1145/361604.361619

ARINC. 1997. *ARINC 651-1: Design Guidance for Integrated Modular Avionics*. ARINC (Aeronautical Radio, Inc).

ARINC. 2013. *ARINC 653P0: Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653*. ARINC (Aeronautical Radio, Inc).

AS 6802. 2011. Time-Triggered Ethernet. SAE International.

N. Audsley, K. Tindell, and A. Burns. 1993. The end of the line for static cyclic scheduling. In *Proceedings of Euromicro Workshop on Real-Time Systems*. 36–41.

Luis Silva Azevedo, David Parker, Martin Walker, Yiannis Papadopoulos, and Rui Esteves Araujo. 2013. Automatic Decomposition of Safety Integrity Levels: Optimization by Tabu Search. In *Workshop on Critical Automotive applications: Robustness and Safety*.

James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russel Urzi. 2009. A Research Agenda for Mixed-Criticality Systems. In *Cyber-Physical Systems Week*.

S.K. Baruah. 2004a. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *Computers, IEEE Transactions on* 53, 6 (2004), 781–784.

Sanjoy Baruah. 2004b. Task Partitioning Upon Heterogeneous Multiprocessor Platforms. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 536–543.

Sanjoy Baruah and Nathan Fisher. 2008. Hybrid-priority Scheduling of Resource-Sharing Sporadic Task Systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*. 248–257.

Sanjoy Baruah and Gerhard Fohler. 2011. Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems. In *Proceedings of the Real-Time Systems Symposium*. 3–12.

Sanjoy Baruah, Haohan Li, and Leen Stougie. 2010. Towards the Design of Certifiable Mixed-criticality Systems. In *Real-Time and Embedded Technology and Applications Symposium*. 13–22.

Sanjoy Baruah and Steve Vestal. 2008. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 147–155.

Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. 2011a. Mixed-Criticality Scheduling of Sporadic Task Systems. In *Annual European Symposium on Algorithms*. 555–566.

S. K. Baruah, A. Burns, and R. I. Davis. 2011b. Response-Time Analysis for Mixed Criticality Systems. In *Proceedings of the Real-Time Systems Symposium*. 34–43.

P. Binns. 2001. A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach. In *Proceedings of the Conference on Digital Avionics Systems*, Vol. 1. 1B6/1–1B6/12.

B. Boehm, C. Abts, and S. Chulani. 2000a. Software development cost estimation approaches–A survey. *Annals of Software Engineering* 10, 1 (2000), 177–205.

Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy, and Bert Steece. 2000b. *Software Cost Estimation with Cocomo II* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Blni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. 2001. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *J. Parallel and Distrib. Comput.* 61, 6 (2001), 810 – 837.

Giorgio Buttazzo. 1997. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston.

L.A. Cortés, P. Eles, and Z. Peng. 2004. Quasi-static scheduling for real-time systems with hard and soft tasks. In *Proceedings of the Conference on Design, automation and test in Europe*. 21176–21181.

Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. 2009. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proceedings of the Real-Time Systems Symposium*. 291–300.

James A. Debardelaben, Vijay K. Madisetti, and Anthony J. Gadient. 1997. Incorporating Cost Modeling in Embedded-System Design. *IEEE Design and Test of Computers* 14 (July 1997),

24–35. Issue 3.
Robert Dick. 2005. Embedded System Synthesis Benchmarks Suite. (2005). http://ziyang.eecs.umich.edu/~dickrp/e3s/
Rolf Ernst. 2010. Certification of Trusted MPSoC Platforms. (2010). 10th International Forum on Embedded MPSoC and Multicore.
M. Gendreau. 2002. *An Introduction to Tabu Search*. Centre for Research on Transportation.
Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. 2013. Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems. In *International Conference on Embedded Software (EMSOFT)*. Montreal, 17:1–17:15.
Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. 2014. Mapping Mixed-Criticality Applications on Multi-Core Architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), Hot-Topic Session on Predictable Multicore Computing*. IEEE, Dresden, Germany, 1–6.
Fred Glover and Manuel Laguna. 1997. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA.
K. Hoyme and K. Driscoll. 1993. SAFEbus. *IEEE Aerospace Electronic Systems Magazine* 8 (1993), 34–39.
IBM. 2010. DO-178B compliance: turn an overhead expense into a competitive advantage. White paper, IBM Rational. (2010). ftp://public.dhe.ibm.com/common/ssi/ecm/en/raw14249usen/RAW14249USEN.PDF
IEC 61508. 2010. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission.
ISO 26262. 2009. ISO 26262 - Road vehicles Functional safety. International Organization for Standardization / Technical Committee 22 (ISO/TC 22).
ISO 9001. 2008. Quality management systems - Requirements. International Organization for Standardization.
V. Izosimov, P. Pop, P. Eles, and Z. Peng. 2008. Scheduling of fault-tolerant embedded systems with soft and hard timing constraints. In *Proceedings of the conference on Design, automation and test in Europe*. 915–920.
M. Jorgensen and M. Shepperd. 2007. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering* 33, 1 (2007), 33–53.
H. Kopetz. 2011a. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer.
H. Kopetz. 2011b. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer.
Yann-Hang Lee, Daeyoung Kim, M. Younis, J. Zhou, and J. McElroy. 2000. Resource scheduling in dependable integrated modular avionics. In *Proceedings of Dependable Systems and Networks*. 14–23.
Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. 2007. A Comparison of Partitioning Operating Systems for Integrated Systems. *Computer Safety, Reliability, and Security* (2007), 342–355.
Haohan Li and Sanjoy Baruah. 2010. An Algorithm for Scheduling Certifiable Mixed-Criticality Sporadic Task Systems. In *Proceedings of the Real-Time Systems Symposium*. 183–192.
Haohan Li and Sanjoy Baruah. 2012. Global Mixed-criticality Scheduling on Multiprocessors. In *Euromicro Conference on Real-Time Systems*. 166–175.
Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. 2010. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *Proceedings of the Conference on Computer and Information Technology*. 1864–1871.
Y. Papadopoulos, M. Walker, M.-O. Reiser, M. Weber, D. Chen, M. Törngren, David Servat, A. Abele, F. Stappert, H. Lonn, L. Berntsson, Rolf Johansson, F. Tagliabo, S. Torchiaro, and Anders Sandberg. 2010. Automatic allocation of safety integrity levels. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness and Safety*. 7–10.

David Parker, Martin Walker, LusSilva Azevedo, Yiannis Papadopoulos, and RuiEsteves Arajo. 2013. Automatic Decomposition and Allocation of Safety Integrity Levels Using a Penalty-Based Genetic Algorithm. In *Recent Trends in Applied Artificial Intelligence*, Moonis Ali, Tibor Bosse, KoenV. Hindriks, Mark Hoogendoorn, CatholijnM. Jonker, and Jan Treur (Eds.). Lecture Notes in Computer Science, Vol. 7906. Springer Berlin Heidelberg, 449–459. DOI:http://dx.doi.org/10.1007/978-3-642-38577-3_46

Paul Pop, Petru Eles, and Zebo Peng. 1999. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings of the International Workshop on Hardware/Software Codesign*. 178–182.

Paul Pop, Petru Eles, and Zebo Peng. 2004. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. Kluwer Academic Publishers.

Paul Pop, Petru Eles, Zebo Peng, Viacheslav Izosimov, Magnus Hellring, and Olof Bridal. 2004. Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications. In *Proceedings of the Conference on Design, automation and test in Europe*. 21028–21033.

P. Pop, P. Eles, Z. Peng, and T. Pop. 2006. Analysis and optimization of distributed real-time embedded systems. *ACM Transactions on Design Automation of Electronic Systems* 11, 3 (2006), 593–625.

P. Pop, V. Izosimov, P. Eles, and Zebo Peng. 2009. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 17, 3 (2009), 389–402.

Paul Pop, Leonidas Tsiopoulos, Sebastian Voss, Oscar Slotosch, Christoph Ficek, Ulrik Nyman, and Alejandra Ruiz Lopez. 2013. Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the RECOMP approach. In *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*.

Traian Pop, Paul Pop, Petru Eles, and Zebo Peng. 2008a. Analysis and Optimisation of Hierarchically Scheduled Multiprocessor Embedded Systems. *International Journal of Parallel Programming* 36, 1 (2008), 37–67.

Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. 2008b. Timing analysis of the FlexRay communication protocol. *Real-Time Systems* 39, 1-3 (2008), 205–235.

Rockwell-Collins. 2009. *Certification Cost Estimates for Future Communication Radio Platforms* (1.1 ed.). Technical Report. Rockwell-Collins.

RTCA DO-178B. 1992. Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA).

John Rushby. 1999. *Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Contractor Report CR-1999-209347. NASA Langley Research Center.

Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. 2010. Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems. *Real-Time and Embedded Technology and Applications Symposium* (2010), 89–98.

Oliver Sinnen. 2006. Fundamental Heuristics. In *Task Scheduling for Parallel Systems*. John Wiley and Sons, Inc., Hoboken, NJ, USA.

Neil R. Storey. 1996. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Domitian Tamas-Selicean, Paul Pop, and Jan Madsen. 2014. *Design of Mixed-Criticality Applications on Distributed Real-Time Systems*. Technical University of Denmark.

Domiţian Tămaş-Selicean and Paul Pop. 2011. Optimization of Time-Partitions for Mixed-Criticality Real-Time Distributed Embedded Systems. In *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. 1–10.

Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. 2012. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proceedings of the International Conference on Hardware/software Codesign and System Synthesis*. 473–482.

J. D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. Syst. Sci.* 10, 3 (1975), 384–393.

Steve Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the Real-Time Systems Symposium*. 239–243.
J. Xu and D. L. Parnas. 1993. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Trans. Softw. Eng.* 19, 1 (1993), 70–84.