Design Optimization of Mixed-Criticality Real-Time Applications on Cost-Constrained Partitioned Architectures

Domiţian Tămaş–Selicean and Paul Pop
DTU Informatics
Technical University of Denmark
Kongens Lyngby, Denmark
Email: dota@imm.dtu.dk, paul.pop@imm.dtu.dk

***Abstract*—In this paper we are interested to implement mixed-criticality hard real-time 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. 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. We are interested to determine (i) the mapping of tasks to processors, (ii) the assignment of tasks to partitions, (iii) the sequence and size of the time slots on each processor and (iv) 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.**

I. 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. There are 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 *separated* (or, *isolated*), so they cannot influence each other. For example, without separation, a lower-criticality task could corrupt the memory of a higher-criticality task.

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 functions are integrated onto the same node. In this context, designers are relying on partitioning mechanisms at the platform level. For example, in the avionics area, the platform-level separation mechanisms are provided by implementations of the ARINC 653 standard, also called "Integrated Modular Avionics" (IMA) [29]. ARINC 653 consists of hardware-mediated operating system-level *spatial* and *temporal* partitioning [29] mechanisms. Similar platform-level separation mechanisms are available in other industries [11], [19].

In this paper we are interested in the design optimization of hard real-time applications with different SILs. We consider heterogenous distributed platforms, consisting of several processing elements (PEs) interconnected using a broadcast bus. 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. However, to simplify the discussion, in this paper, we assume that all applications are scheduled using static-cyclic scheduling (SCS). In [33] we have shown how applications scheduled using a fixed-priority preemptive scheduling (FPS) policy can be handled in a partitioned architecture. 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 [1], 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 [14] 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 [17] by enforcing a Time-Division Multiple Access scheme. Researchers have shown how realistic bus protocols such as TTP [22] and FlexRay [28] 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 [17], or software-based solutions such as re-execution, replication and checkpointing [25]. 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.

A. Contribution

In this paper we are interested to implement mixed-criticality hard 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 sequence and size of the partition time slots on each PE and (iv) the schedule tables for all PEs. This is the first time, to our knowledge, that such a problem has been addressed. We propose a Tabu Search (TS)-based approach for this design optimization problem.

In [33] 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.

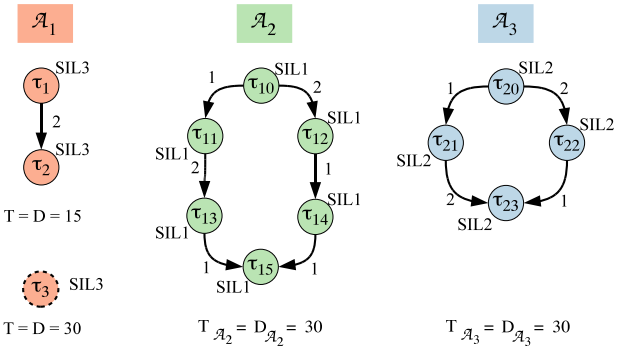
Therefore, in this paper we address the case when the *sharing* of partitions by tasks from applications with different SILs is allowed, aiming at integrating more applications onto a given platform, without increasing unit costs. If tasks of different SILs share a partition, they will have to be developed 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 by paying with increased development costs, instead of increased unit costs.

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

II. Application Model

The set of all applications in the system is denoted with Γ. We model an application as a directed, acyclic graph $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \Gamma$. The graph is polar, which means that there is a *source node*, which is a node that has no predecessors and a *sink node* that has no successors. 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) $C_i^{N_j}$ for 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. We assume that the message sizes s_{m_i} of each message m_i are known. 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 graph $\mathcal{G}_i$.



(a) Example mixed-criticality applications

	$\mathcal{A}_1$			$\mathcal{A}_2$						$\mathcal{A}_3$			
	τ_1	τ_2	τ_3	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{20}	τ_{21}	τ_{22}	τ_{23}
N_1	x	3	x	2	3	x	2	6	2	4	6	4	x
N_2	4	5	3	3	x	6	3	9	6	9	10	5	4

(b) WCET and mapping restrictions

	$\mathcal{A}_1$			$\mathcal{A}_2$						$\mathcal{A}_3$			
	τ_1	τ_2	τ_3	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{20}	τ_{21}	τ_{22}	τ_{23}
SIL 1	x	x	x	2	3	3	4	3	4	x	x	x	x
SIL 2	x	x	x	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)

Figure 1: Application model example

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. An "x" 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.

If dependent tasks are of different periods, they are combined into a merged graph capturing all activations for the hyper-period (LCM of all periods). Release times of some tasks as well as multiple deadlines can be also be modeled [22].

A. Safety Integrity Levels

As mentioned, a *safety-critical* system should not endanger human life or the environment. A *hazard* is a situation in which there is actual or potential danger to people or to the environment. *Risk* is a combination of the frequency or probability of a specified hazardous event, and its consequence. If, after performing an initial hazard and risk analysis, a system is deemed safety-related, it has to be certified [32]. Certification is a "conformity of assessment" performed by a third party. The current certification practice is "standards-based" [30], and re-

quires that prescribed certification standards are followed, depending on the application area. For example, IEC 61508 is used in industrial applications, ISO 26262 is for the automotive area, whereas DO-178B refers to software for airborne systems.

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. SIL allocation is typically a manual process, which is done after performing hazard and risk analysis [32], but researchers have proposed automatic approaches for SIL allocation [20]. Although SILs differ slightly among areas (for example, the avionics area uses five "Design Assurance Levels" (DAL), from DAL A to DAL E), the approach presented in this paper is applicable to all safety-critical areas, regardless of the standard. SILs are assigned to safety functions, from SIL 4 (most critical) to SIL 0 (non-critical). Functions are decomposed into tasks. We introduce the notation $SIL : \mathcal{V}_i \rightarrow \{SIL\ k\}$, 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.

B. Development Cost Model

The SIL assigned to a task will dictate the development processes and certification procedures that have to be followed. SIL 0 functions are non-critical and can be developed using any methods. For SIL 1, a more systematic approach is needed, to the level required by quality management standards such as ISO 9001. SIL 2 is quite similar to SIL 1, but typically involves more reviewing and testing. SIL 3 is significantly more difficult. Certification standards will suggest specific methods to be followed, and provide a checklist of techniques that are recommended to be applied. If "semi-formal" methods are acceptable in lower SILs, SIL 4 often requires formal methods, increasing further the difficulty and development costs associated to building safety-critical systems.

Software development cost estimation is a widely researched topic, and is beyond the scope of this paper. The reader is directed to [16], [4] for reviews on this topic. One of the most influential software cost models is the Constructive Cost Model (COCOMO) [5]. Researchers have shown how to take into account the development costs during the design process of embedded systems [9]. The development of safety-critical systems is a highly structured and systematic process dictated by standards. Hence, we believe that is reasonable to assume that the designer will use a cost model to capture the development costs associated to a given SIL.

Thus, 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 with $DC(\mathcal{A}_i)$, 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.

C. Separation Requirements

Tasks of different SILs have to be separated. Otherwise, for example, a lower-criticality task could write in the code or data area of a higher-criticality task, leading thus to a failure. Separation 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 of different SILs.

Standard practice in certain areas may place additional separation requirements. For example, it may be recommended that two tasks of SIL 4 from different applications should be separated, although they are at the same SIL level. To capture such requirements, and any additional separation requirements desired by the designer, we define the separation requirements graph $\Pi(\mathcal{V}, \mathcal{E})$ as a bidirectional graph. $\mathcal{V}$ represents the set of all tasks, while $\mathcal{E}$ is the set of edges. An edge $sr_{ij} \in \mathcal{E}$ means that tasks τ_i and τ_j are not allowed to share a partition.

III. System Model

We consider architectures composed of a set $\mathcal{N}$ of PEs which share a broadcast communication channel. In this paper we use a simple statically scheduled bus, where the communication takes place according to a static schedule table computed offline. Also, all applications are scheduled using non-preemptive static-cyclic scheduling.

A. Separation through Partitioning

If two tasks are of different SILs, or if they have to be separated according to the separation requirements graph Π, we consider that the separation is achieved through partitioning. 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 $\mathcal{P}_{ij}$ of k partition slices p_{ij}^k, $k \geq 1$. A partition slice p_{ij}^k 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 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 in a MF are the same (on a given PE), but the contents of the slices can differ.

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,

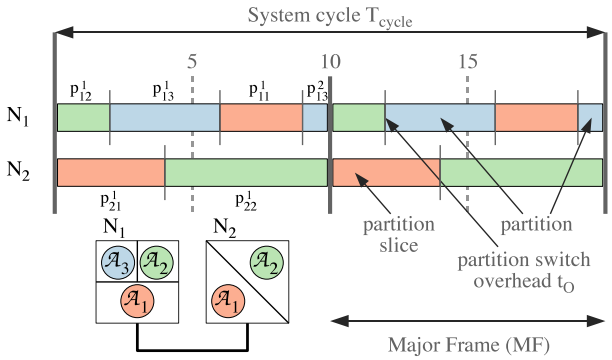


Figure 2: Partitioned architecture

the tasks of $\mathcal{A}_3$, for example, can execute only in partition P_3 on PE N_1, composed of the sequence $P_{1,3}$ of partition slices $p^1_{1,3}$ and $p^2_{1,3}$. In this example, all the tasks of $\mathcal{A}_3$ have the same SIL. However, tasks of different SILs in an application have to be in separate partitions. Another example of partitioning is presented in Fig. 3c, where we have the 3 applications from Fig.1a implemented on 2 PEs, with $T_{MF} = 15$ and $T_{cycle} = 30$.

The schedule tables $\mathcal{S}$ 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_{14} in Fig. 3c, which shows the schedule tables for the applications in Fig.1a. The time overhead due to partition switching is denoted with t_O, and our optimization approach takes into account the partition switching overheads.

B. Elevation and Software-based Separation

Partitioning introduces overheads because it constrains the way tasks can use the PEs, leading to unused slack inside certain partition slices. Our goal in this paper is to optimize the mapping and partitions such that these overheads are minimized, increasing thus the chance to find schedulable implementations.

However, there might be situations when finding schedulable implementations is possible only if we allow *sharing* of partitions by tasks with different SILs. This would help to further reduce the unused slack, and thus allow us to integrate more applications onto a given cost-sensitive platform. Note that two tasks can share a partition only if they have the same SIL (and are not required by Π to be separated). However, to a task of a lower SIL can always be assigned a higher SIL, i.e., it can be *elevated*. Tasks elevated to the same SIL can then share a partition.

This will not only increase the development costs for the elevated task, but it may trigger the elevation of other tasks. As mentioned in Section II-C, 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 recursively the elevation of its predecessors. This, in turn, can trigger the elevation of other tasks, if such predecessors will thus have a higher SIL in another partition slice. For example, considering the application details from Fig. 1a, in Fig. 3d task τ_3 shares the partition with tasks τ_{13} and τ_{12}. The two tasks have a lower SIL than τ_3, and as such they have to be elevated to SIL 3. This in turn triggers the need to elevate all of τ_{12}'s and τ_{13}'s predecessors, namely τ_{10} and τ_{11}. Since tasks τ_{14} and τ_{15}, share the partition with τ_{10} and τ_{11}, they in turn need to be elevated to SIL 3. Thus, all the tasks of application $\mathcal{A}_2$ have been elevated from SIL 1 to SIL 3.

Partition sharing of tasks with different SILs may also be possible if *software-based separation* mechanisms are employed. Such separation mechanisms[1] are typically used between lower SIL tasks (e.g., SIL 1 and SIL 2). Spatial separation can be obtained using methods such as *Software Fault Isolation* (SFI) [29], or compiler and linker mechanisms [12]. Temporal separation can rely on watchdogs. For simplicity, we have decided to classify such methods as "elevation", since they are conceptually similar: paying with increased development costs to attain separation.

IV. Problem Formulation

The problem we are addressing in this paper can be formulated as follows: given (1) a set Γ of applications, (2) the criticality level $SIL(\tau_i)$ of each task τ_i, (3) the separation requirements Π between the tasks, (4) an architecture consisting of a set $\mathcal{N}$ of processing elements, (5) the size of the major frame T_{MF} and (6) 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 mapping M of tasks to PEs, (2) the set $\mathcal{P}$ of partition slices on each processor, including their order and size, (3) the assignment ϕ of tasks to partitions and (4) the schedule $\mathcal{S}$ for all the tasks in the system.

A. 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_3, 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 IV-B.

Let us first consider the case when mapping and partitioning optimization are performed separately. Thus, Fig. 3a presents the mapping and schedules for the case when there is no partitioning, 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 the cost function presented in Eq. 2 (Section VI), which tries to minimize the schedule lengths of the applications.

[1]Such separation may introduce additional performance overheads, e.g., may require runtime checks of memory accesses. Our model can easily be extended to capture these overheads.

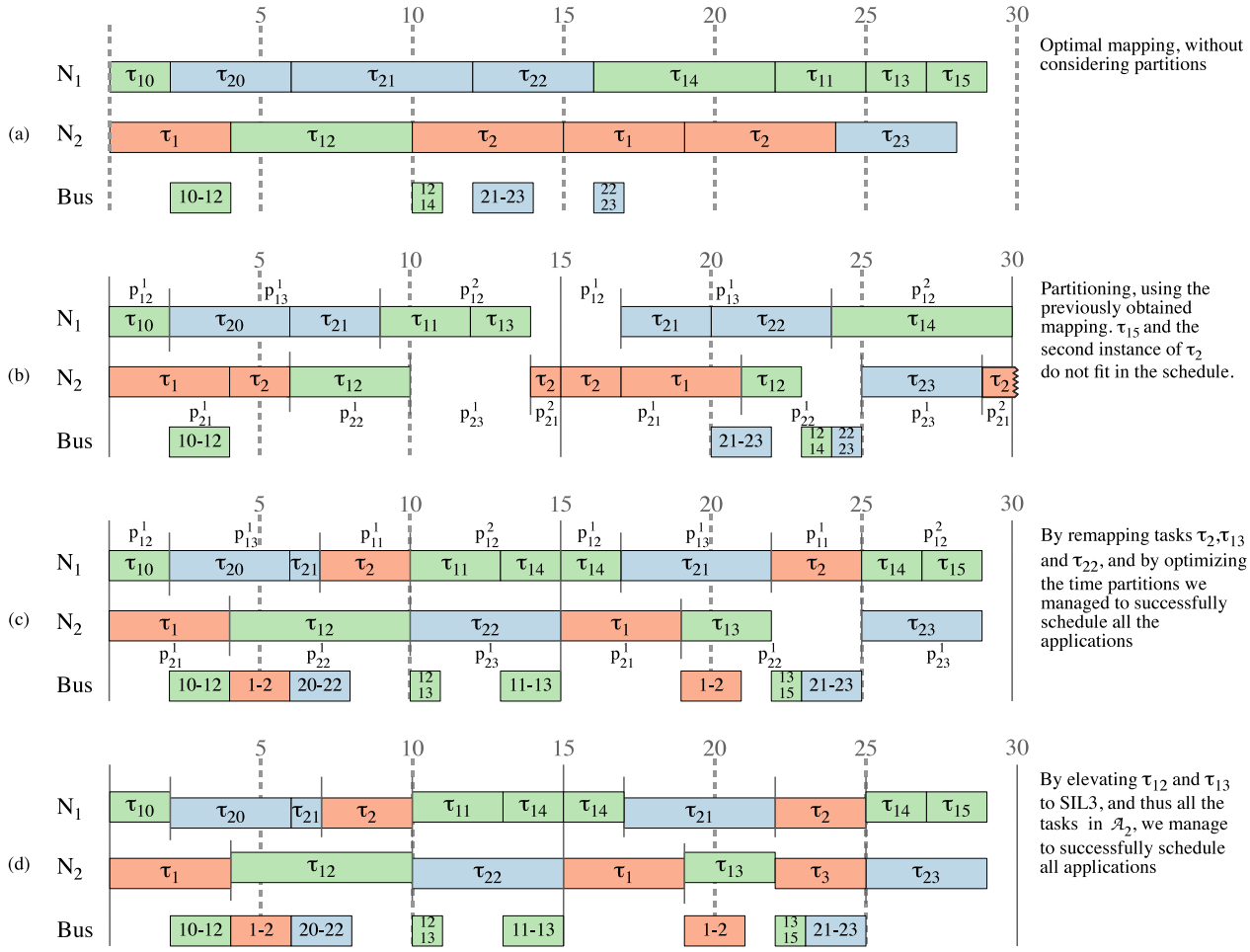


Figure 3: Motivational example

In Fig. 3a we show the schedules on each resource, 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, "10–12" is sent from task τ_{10} to τ_{12}. The dashed vertical lines are timeline guides, and should not be interpreted as partitions.

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 Eq. 2, which drives the optimization towards schedulable solutions), 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_{10} and τ_{20} represent partition slice boundaries. The partition slices are denoted with the notation p_{ij}^k introduced in Section III-A.

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_{15} and the second instance of task τ_2 do not fit into the schedule, and thus applications $\mathcal{A}_1$ and $\mathcal{A}_2$ are not schedulable.

In this paper we consider that the optimization of mapping and partitioning is done 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_{13} and τ_{22} from N_1 to N_2 and of task τ_2 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.

B. 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_3 to application $\mathcal{A}_1$, see Fig. 1a. In this case, we are unable to obtain a schedulable implementation. Note that, although it may seem that task τ_3 would fit in-between tasks τ_{13} and τ_{23} in the schedule of N_2 in Fig. 3c, τ_3, which is SIL 3, cannot use that partition, which is for SIL 1 tasks.

Moreover, the particular partition slice cannot be split, because then it would not fit task τ_{12} 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_3 of SIL 3 to share the partition with tasks τ_{12} and τ_{13} of SIL 1, by elevating these two tasks to SIL 3. This will trigger the elevation of the predecessors of τ_{12} and τ_{13}, namely τ_{10} and τ_{11}, to SIL 3. In addition, since τ_{10} and τ_{11} share partitions with tasks τ_{14} and τ_{15}, 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 85,000 Euros to 127,000 Euros. The solution in Fig. 3d is schedulable, and is optimal in terms of development costs as captured by the cost function from Eq. 1 discussed in Section V-A.

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.

V. Tabu Search-based Design Optimization

The problem presented in the previous section is NP-complete [34]. In order to solve this problem, we will use the "Mixed-Criticality Design Optimization" (MCDO) strategy from Fig. 4. MCDO takes as input a set of applications Γ (including the SIL information, development costs DC and the separation requirements graph Π) and the set of processing elements $\mathcal{N}$, and returns the implementation Ψ consisting of 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 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. 4. The initial mapping M° is done such that the utilization of processors is balanced and the communication on the bus is minimized. $\mathcal{P}^\circ$ 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 V-A) to determine 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 task mapping M, the optimized partitions $\mathcal{P}$ and the assignment ϕ of tasks to partitions obtained in line 2 in Fig. 4, we use a List Scheduling heuristic (see Section V-B) to determine the schedule tables for the applications.

A. Tabu Search

Tabu Search (TS) [13] is a meta-heuristic optimization, which searches for that solution which minimizes the *cost function*. Tabu Search takes as input the set of applications Γ, the set of PEs $\mathcal{N}$, and the initial solution, consisting of M°, $\mathcal{P}^\circ$, and ϕ°, and returns at the output the best solution Ψ found during the design space exploration, in terms of the cost function. We define the cost function of an implementation Ψ as:

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

R_i is the response time of the application, while D_i is the deadline of the application. For each alternative solution visited by TS we use the List Scheduling-based heuristic from Section V-B 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 II-B). 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 the applications are schedulable, we are interested to minimized the development cost.

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"), to prevent the search from revisiting previous solutions, thus avoiding cycling. If the currently explored solution is better than the best known solution, it is saved as the "best-so-far" solution and, to prevent cycling, the move that created this solution is saved as "Tabu". In case there is no improvement in finding a better solution for a number of iterations, we use *diversification*, i.e., we visit previously unexplored regions of the search space. In case the search diversification is unsuccessful, we *restart* the search from the best known solution.

We use one *re-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*. The task *re-assignment* move re-assigns a task to another partition. Randomly chosen by the algorithm,

MCDO(Γ, $\mathcal{N}$)

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

Figure 4: Mixed-Criticality Design Optimization strategy

the partition can be either an existing one, or a newly created one. The partition can be on another PE, thus, implicitly, the re-assignment move will also *re-map* the task. The re-assignment move respects the separation requirements graph Π, but does not prevent partition sharing by tasks of different SILs. In case the move will lead to sharing, we recursively elevate tasks as required, and update the development costs accordingly. In case the partition the task is moved from, has no other tasks assigned to it, it is deleted and the processing time is distributed to a randomly chosen partition. As a result, the algorithm creates and deletes partitions on the fly as needed, depending on the task *re-assignment* moves.

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 partition can be resized is randomly chosen, but we have imposed an upper limit (half the size of the partition). 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. 5 illustrates how Tabu Search works. We consider applications $\mathcal{A}_1$ and $\mathcal{A}_3$ from Fig. 1, with periods and deadlines equal to 16. The size of the major frame T_{MF} is set to 8 and the T_{cycle} is 16. The current solution is presented in Fig. 5a, which is also the best-so-far solution. Note that this solution is not schedulable, since tasks τ_{21} and τ_{23} from $\mathcal{A}_3$ do not fit into the schedules. Several neighbor solutions generated starting from Fig. 5a, are presented in Fig. 5b–5e, and are intended to illustrate the types of moves performed by TS and how TS updates the "Tabu history". 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 type and details of the move, the "Tabu history" and the cost function. For the partition moves, the tabu attributes are the PEs, whereas in the case of the task re-assignment moves, the tabu attributes are the applications. The move in Fig. 5d is Tabu, and it is not better than the current solution, hence it is removed from the set of solutions to be explored. Fig. 5e presents a re-assignment move which delivers a solution that is Tabu, but it is better than the best-so-far solution. Note that, because of the re-assignment of τ_2 to the partition of $\mathcal{A}_3$ on N_1, the partition assigned to $\mathcal{A}_1$ is deleted and the time is given to the other partition. Also, tasks τ_{20} and τ_{21} have to be elevated to SIL 3, since τ_2 is SIL 3. The Tabu history is updated as shown next to Fig. 5e and the iterations continue with this solution as the current solution.

B. List Scheduling

The applications are scheduled using static-cycling non-preemptive scheduling. Given a mapping $\mathcal{M}$, a partition set $\mathcal{P}$ and the assignment ϕ of tasks to partitions, we

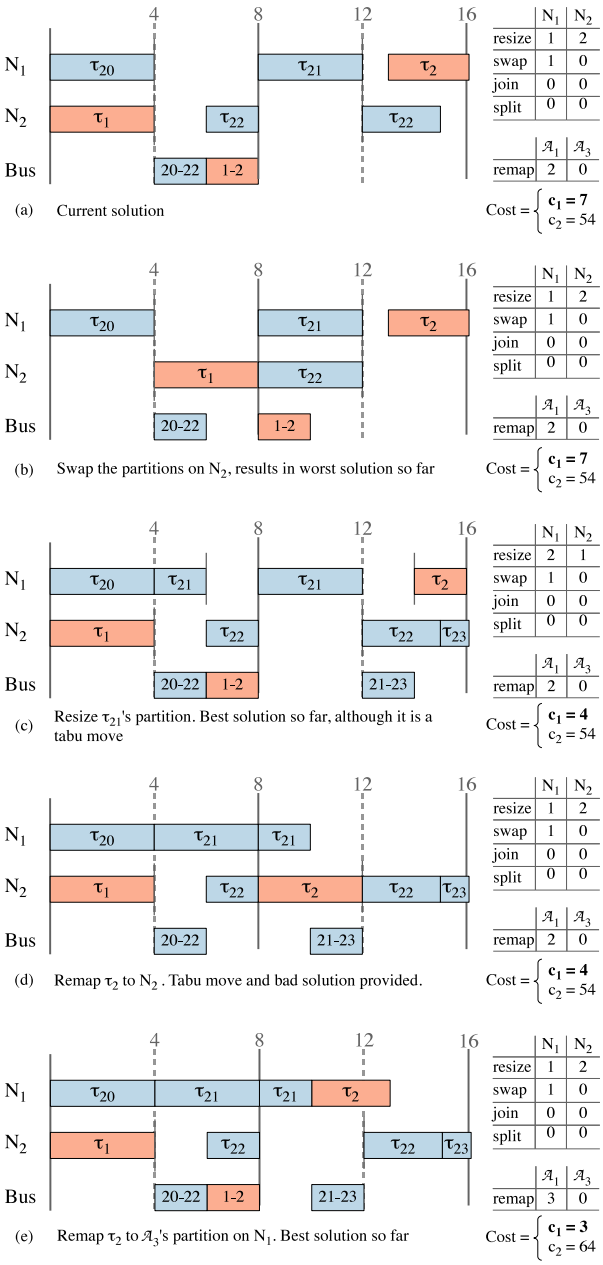


Figure 5: Moves and tabu history

use a List Scheduling (LS)-based heuristic to determine the schedule tables $\mathcal{S}$ for each application. 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 [22] 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 $\mathcal{P}$. If a partition slice finishes before a task has completed its execution (as is the case with $\tau_{21} \in \mathcal{A}_3$ in Fig. 3d), we assume that the task is suspended and will continue its execution in the next partition where is assigned. Our LS implementation takes into account the partition switching overhead t_O. The suspension of

the task will take place online, based on the partition scheme $\mathcal{P}$ loaded into the kernel and t_O contains the time needed to do a context switch to another partition. LS also schedules the messages on the bus.

VI. Experimental Evaluation

For the evaluation of our proposed algorithm "Mixed-Criticality Design Optimization" (MCDO) approach we used 7 synthetic benchmarks and 3 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 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.

We were interested to compare the number of schedulable implementations found by MCDO with two other setups: (i) when the sharing of partitions by tasks of different criticality levels is not allowed, but mapping and partitioning optimization (MPO) is performed simultaneously. In the second setup, (ii) sharing is 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. 4, 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, and MPO does not allow reassignment moves that would lead to partition sharing by mixed-criticality tasks. Also, MO, PO and MPO use a slightly different cost function (compared to Eq. 1), where we do not consider development costs (the term c_2), which are constant since we do not elevate tasks to higher SIL levels:

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

where now the term c_2 is used when the applications are schedulable and captures the "degree of schedulability" of an implementation. To have a fair comparison, we have used time limits corresponding to the size of the design space. Thus, MO+PO has a time limit of 30 minutes, MPO uses a time limit of 60, while MCDO runs for 480 minutes.

The three strategies, MO+PO, MPO and MCDO correspond to Fig. 3b, Fig. 3c and Fig. 3d, respectively, in the motivational example discussed in Section IV. The results for the first set of experiments are presented in Table I in rows 2-6. 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, in Table I.

As we can see from the comparison between MO+PO and MPO, there is a significant improvements 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 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 in the same run, is able to schedule 3 out 4 applications.

If MPO produces a schedulable solution, i.e., the applications are schedulable without using 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, using elevation to allow partition sharing can find schedulable implementations in all cases. There are situations where MCDO is able to find schedulable implementations using partition sharing, but without the need of elevating tasks (the tasks have the same criticality level). Such a situation is in line 2 and in line 11 in the table, where the zero development cost means that the solution was produced without using elevation.

Once a schedulable implementation is found by using elevation, the cost function from Eq. 1 will drive MCDO to solutions that minimize the development cost. The increase in development cost that we have to pay in order to find schedulable implementations, compared to MPO which does not perform SIL elevation, is reported in the last column of Table I.

We have also compared MPO to MO+PO in terms of the cost function. The percentage improvement in the cost function, i.e., the "degree of schedulability" is reported in column 8. An increase in the "degree of schedulability", in the case of a schedulable implementation, as is the case for the third test case, means that it is possible to implement the solution on a slower (cheaper) architecture.

In the second set of experiments, labeled "Set 2" in Table I, we were interested to see how MCDO performs compared to MO+PO and MPO as the utilization of the system increases. Thus, we have mapped the number of mixed-criticality applications from 3 to 6, but we have used the same architecture of 4 PEs. As we can see, for the smaller benchmarks of 3 and 4 applications, MO+PO is able to find schedulable implementations. Optimizing the mapping and time partitions using MPO leads to more schedulable implementations. However, as the system utilization increases, as is the case for the largest benchmark in this set, where we used 6 applications on 4 PEs, only MCDO, which considers elevation to allow partition sharing by tasks of mixed-criticality, is able to provide schedulable solutions.

Finally, we have also used 3 real life benchmarks derived from the Embedded Systems Synthesis Benchmarks Suite (E3S) version 0.9 [10]. We have used the *consumer-cords*, *networking-cords* and *telecom-cords* benchmarks.

Table I: Comparison of MO+PO, MPO and MCDO

Set	Test Case	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	450.00	All	0
	2	4	34	4	0	3	3600.00	All	99
	3	5	41	5	3	All	235.00	–	–
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	835
3	consumer	2	12	3	0	1	343.45	All	25
	networking	4	13	3	2	2	31.78	All	15
	telecom	9	30	3	5	8	8915.09	All	0

In all three cases we were interested to implement the applications to an architecture of 3 PEs. The results obtained from these real-life benchmarks are reported in the last 3 lines in Table I and confirm the results of the synthetic benchmarks.

VII. Related Work

There is a large amount of research on hard real-time systems [6], [17], including task mapping to heterogeneous architectures [26]. 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. However, there is little research work on the integration of mixed *safety*-criticality applications onto the same platform.

In the context of mixed TT/ET systems, Pop et al. [27] 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 [24]. While in [27] and [24] TT and ET tasks share the same processor, the work in [23] 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 [7] (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 in [15] 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) [1], 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 in [31], 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 [21] and a mixed TT/ET bus such as FlexRay [28] 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. [18] 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 [3], with the difference that FPS is used for scheduling both the partitions (which are normally scheduled using SCS) and the tasks. Binns [3] 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.

Our work allows tasks with different criticality levels to share a partition only if the lower-criticality tasks are elevated at the higher-criticality level. Current certification practice requires separation, and can only remove such a requirement if the two tasks are at the same criticality level.

There are several works where mixed-criticality tasks are addressed. For example, Baruah et al. [2] propose a task model that can capture mixed-criticality functions, together with an associated schedulability analysis. Niz et al. [8] discuss the issue of "criticality inversion", similar to the classical priority inversion problem, and propose a "zero-slack scheduling" scheme for such a context. However, this work assumes that tasks of different criticality share the same processor with little or no separation (i.e., there is no spatial-partitioning). Today, this practically means that all the tasks are developed and certified at the highest criticality level, which is not feasible, due to the prohibitive development and certification costs. Such research assumes that in the future the certification practice

will change, to allow different criticality tasks to share the same platform. For example, a vision of "just-in-time certification" [30] is proposed by Rushby. However, the current standards-based certification practice is unlikely to change in the near future.

VIII. 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, or separation using software-based mechanisms. Both approaches to sharing lead to increased development costs. Our optimization approach finds that schedulable implementation on a cost-constrained architecture, which minimizes the development costs.

Acknowledgements

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.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of Real-Time Systems Symposium*, pages 4 –13, 1998.

[2] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symp.*, 2010.

[3] P. Binns. A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach. In *Conf. on Digital Avionics Systems*, volume 1, pages 1B6/1 –1B6/12, 2001.

[4] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approachesA survey. *Annals of Software Engineering*, 10(1):177–205, 2000.

[5] B. Boehm, R. Madachy, and B. Steece. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, USA, 2000.

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

[7] L. Cortés, P. Eles, and Z. Peng. Quasi-static scheduling for real-time systems with hard and soft tasks. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, page 21176. IEEE Computer Society, 2004.

[8] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proc. of the Real-Time Systems Symposium*, pages 291–300, 2009.

[9] J. A. Debardelaben, V. K. Madisetti, and A. J. Gadient. Incorporating cost modeling in embedded-system design. *IEEE Des. Test*, 14:24–35, July 1997.

[10] R. Dick. Embedded system synthesis benchmarks suite. http://ziyang.eecs.umich.edu/dickrp/e3s/.

[11] R. Ernst. Certification of trusted mpsoc platforms. 10th International Forum on Embedded MPSoC and Multicore, 2010.

[12] EUROCAE. ED-94B - Final annual report for clarification of ED-12B. Technical report, EUROCAE.

[13] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[14] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace Electronic Systems Magazine*, 8:34–39, 1993.

[15] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Scheduling of fault-tolerant embedded systems with soft and hard timing constraints. In *Proceedings of the conference on Design, automation and test in Europe*, pages 915–920. ACM, 2008.

[16] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *Software Engineering, IEEE Transactions on*, 33(1):33–53, 2007.

[17] H. Kopetz. *Real-Time Systems-Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[18] Y.-H. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Proc. of Dependable Systems and Networks*, pages 14 –23, 2000.

[19] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. A Comparison of Partitioning Operating Systems for Integrated Systems. *Computer Safety, Reliability, and Security*, pages 342–355, 2007.

[20] Y. Papadopoulos et al. Automatic allocation of safety integrity levels. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 7–10. ACM, 2010.

[21] P. Pop, P. Eles, and Z. Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proc. of the Workshop on Hardware/software Codesign*, pages 178–182, 1999.

[22] P. Pop, P. Eles, and Z. Peng. *Analysis and Synthesis of Communication-Intensive Heterogenous Real-Time Systems*. Kluwer Academic Publishers, 2004.

[23] P. Pop, P. Eles, Z. Peng, V. Izosimov, M. Hellring, and O. Bridal. Design optimization of multi-cluster embedded systems for real-time applications. 2004.

[24] P. Pop, P. Eles, Z. Peng, and T. Pop. Analysis and optimization of distributed real-time embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2006.

[25] P. Pop, V. Izosimov, P. Eles, and Z. Peng. 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):389 –402, march 2009.

[26] R. Pop. *Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-based Interconnections*. PhD thesis, Linköping University, 2011.

[27] T. Pop, P. Pop, P. Eles, and Z. Peng. Analysis and optimisation of hierarchically scheduled multiprocessor embedded systems. *Intl. Journal of Parallel Programming*, 2008.

[28] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1-3):205–235, 2008.

[29] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[30] J. Rushby. Just-in-time certification. 2007.

[31] P. K. Saraswat, P. Pop, and J. Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:89–98, 2010.

[32] N. Storey. *Safety critical computer systems*. Addison-Wesley Longman Publ. Co., Inc. Boston, MA, USA, 1996.

[33] D. Tamas-Selicean and P. Pop. Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops , IEEE International Symposium on*, 0:1–10, 2011.

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