2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops

Optimization of Time-Partitions for Mixed-Criticality Real-Time Distributed Embedded Systems

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

***Abstract*—In this paper we are interested in mixed-criticality embedded real-time applications mapped on distributed heterogeneous architectures. The architecture provides both spatial and temporal partitioning, thus enforcing enough separation for the critical applications. With temporal partitioning, each application is allowed to run only within predefined time slots, allocated on each processor. The sequence of time slots for all the applications on a processor are grouped within a Major Frame, which is repeated periodically. We assume that the safety-critical applications (on all criticality levels) are scheduled using static-cyclic scheduling and the non-critical applications are scheduled using fixed-priority preemptive scheduling. We consider that each application runs in a separate partition, and each partition is allocated several time slots on the processors where the application is mapped. We are interested to determine the sequence and size of the time slots within the Major Frame on each processor such that both the safety-critical and non-critical applications are schedulable. We have proposed a Simulated Annealing-based approach to solve this optimization problem. The proposed algorithm has been evaluated using several synthetic and real-life benchmarks.**

***Keywords*-mixed-criticality; real-time systems; temporal-partitioning**

I. Introduction

Depending on the particular application, an embedded system has certain requirements on performance, cost, dependability, size etc. In a *hard real-time embedded system* the "correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced" [1]. *Safety* is a property of a system that will not endanger human life or the environment. *Safety-Integrity Levels* (SILs) capture the required protection against failure when building a safety-critical embedded system, and will dictate the development processes and certification procedures that have to be followed. There are four SIL levels, ranging from 4 (most critical) to 1 (least critical).

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. The application software running on such distributed architectures is composed of several functions. The way the functions have been distributed on the architecture has evolved over time. Initially, in automotive and aerospace applications, for example, each function was running on a dedicated hardware node, allowing the system integrators to purchase nodes implementing required functions from different vendors, and to integrate them together into their system (this approach is also called a "federated architecture"). However, the number of such nodes in the architecture has exploded, reaching over one hundred in an airplane or a high-end car, leading to increased wiring, increased costs, size, weight and power consumption.

These trends have created a huge pressure to reduce the number of nodes, use the resources available more efficiently, and thus reduce costs. This is achieved through the integration of several functions in one node (also called an "integrated architecture"). The same trends are driving the integration of several levels of safety-criticality onto the same node, together with non safety-critical functions. The "Research Agenda for Mixed-Criticality Systems" [2] 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". The current practice to mixed-criticality systems is to physically separate the different criticality functions in different hardware components, so they cannot influence each other.

In avionics, the proposed integration solution is based on "Integrated Modular Avionics" (IMA) [3], which allows the integration of mixed-criticality functions onto the same node as long as there is enough *spatial* and *temporal* partitioning [3]. A similar problem is faced in many other industries. This is coupled with the trend towards using multi-core systems, where several processors can be integrated onto a single chip, decreasing the costs, power consumption, size, and increasing the performance through parallelization [4]. In Europe, multi-cores are addressed through EU projects such as RECOMP ("Reduced certification cost for trusted multi-core platforms" [4]), which has the goal to "define a European standard reference technology for mixed-criticality multi-core systems supported by the European tool vendors".

In this paper we consider heterogeneous distributed embedded systems that have to implement hard real-time applications with different Safety-Integrity Levels (SILs), including non-critical functions. We assume that the hardware and software architecture provides both spatial and temporal partitioning, thus enforcing enough separation for the mixed-criticality applications. The system architecture is presented in Section III, which includes a discussion on partitioning. Such partitioned architectures are available not only in avionics (where IMA is used), but will also be available in other application areas [4]. We consider that the mapping of tasks to the processors is given.

There are two basic approaches for handling tasks in real-time applications [1]. In the event-triggered approach (ET), activities are initiated whenever a particular event is noted. In the time-triggered (TT) approach, activities are initiated at predetermined points in time. There has been a long debate in the real-time and embedded systems communities concerning the advantages of TT and ET approaches [5], [1]. Safety-critical applications are typically designed using a TT approach to ensure both the predictability of worst-case behavior, and high resource utilization [1]. One disadvantage of the TT approaches is their lack of flexibility, which is an important aspect in non-critical systems.

Therefore, in this paper, we assume that the safety-critical (SC) applications are scheduled using static-cyclic scheduling (SCS) and the non-critical (NC) applications are scheduled using a fixed-priority preemptive scheduling (FPS) policy. For the bus we use the *Universal Communication Model* [6], which can model both TT and ET traffic. Section II presents the application model. Note that our model, based on [7], is more general and allows any assignment of scheduling policies to applications (i.e., FPS for SC applications), including a combination of TT and ET tasks within the same application. Moreover, although we assume that NC applications are hard real-time, soft real-time functions can also be integrated with fixed-priority preemptive scheduling using a technique such as the Constant Bandwidth Server (CBS) [8], where the server is seen as a hard task providing a desired level of service to soft tasks, issue discussed by us in [9].

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 [1], or software-based solutions such as re-execution, replication and checkpointing. In [10] we have shown how checkpointing and active replication can be combined in an optimized implementation that leads to a schedulable fault-tolerant application without increasing the amount of employed resources. Hence, 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 SC applications such that they provide the required level of fault-tolerance.

A. Contribution

In this paper we consider that each application runs in a separate partition, and we are interested to determine the sequence and length of the time partitions on each processor such that both the SC and NC applications are schedulable. The exact problem formulation is presented in Section IV and illustrated in Fig. 1. This is the first time, to our knowledge, that such a problem has been addressed.

We have proposed a Simulated Annealing-based approach (SA) to solve this optimization problem, which is presented in Section V, and evaluated using several benchmarks in Section VI. SA decides the sequence (order) of partitions and their length, for each processor.

For each tentative solution produced by SA (consisting of the sequence and length of partition slices) we determine, as follows, if the deadlines are satisfied and how much slack is available (for future upgrades, for example). We first run a *List Scheduling*-based algorithm (Section V-B) for the SCS applications, to obtain the schedule tables within the defined partitions, and then a *Response-Time Analysis* (Section V-C) for the FPS applications, which we have extended to obtain the worst-case response times considering the interruption from other time-partitions.

This our is first attempt to solve the problem of optimizing the time-partitions, hence we have decided to use SA because of its simplicity, although, in our experience, a Tabu Search meta-heuristic may produce better results. The right choice of optimization approach depends on the particular problem, and we plan to investigate which is the best approach in our future work. Note that an approach based on Integer Linear Programming can also be used to determine the partitions, but it may be difficult to integrate with the schedulability analysis.

B. Related Work

Lee et al. [11] 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 [12], with the difference that FPS is used for scheduling both the partitions (which are normally scheduled using SCS) and the tasks. Binns [12] 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.

The problem of the optimization of time-partitions has been addressed at the bus level, but without considering partitions at the processor level, mixed-criticality applications and different scheduling policies. Researchers have shown how a Time-Division Multiple Access bus such as the TTP [13] and a mixed TT/ET bus such as FlexRay [14] can be optimized to decrease the end-to-end delays.

In the context of mixed TT/ET non-critical systems, Pop et al. [7] 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.

There are several works where mixed-criticality tasks are addressed, but researchers assume that tasks of different criticality share the same processor with little or no separation (i.e., there is no spatial-partitioning). Baruah et al. [15] propose a task model that can capture mixed-criticality functions, together with an associated schedulability analysis. Niz et al. [16] discuss the issue of "criticality inversion", similar to the classical priority inversion problem, and propose a "zero-slack scheduling" scheme for such a context.

We have addressed the optimization of CBS-server capacity in the context of mixed hard and soft real-time tasks [9], such that the hard tasks are schedulable and the quality of service for the soft tasks is maximized. CBS-servers provide a time-partitioning between hard and soft tasks.

II. Application Model

The set of all applications in the system is denoted with Γ. A function $F : \Gamma \rightarrow \{SC, NC\}$ captures if an application $\mathcal{A}_i \in \Gamma$ is safety-critical (SC) or non-critical (NC). We do not distinguish among different criticality levels for the safety-critical applications. We assume that each SC application has been developed according to the certification requirements for the particular SIL. We model an application as a directed, acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. 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 $\tau_i \in \mathcal{V}$ represents one task.

The tasks are mapped on a distributed heterogeneous architecture, described in the next section. The mapping of each task τ_i is denoted by the function $M : \mathcal{A} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of processing elements (PEs) in the architecture. Each task τ_i is characterized by a worst-case execution time C_i (on the PE that is assigned to for execution).

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. An edge $e_{ij} \in \mathcal{E}$ from τ_i to τ_j indicates a synchronous communication: τ_j waits for the output of τ_i. Tasks can also communicate asynchronously through buffers, i.e., a reader task will block if the buffer is empty and a writer task will block if the buffer is full. In this case, we assume that the buffer sizes have been determined such that there is no overflow or underflow. Such a communication is not captured explicitly in our model.

For SCS applications, 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 task graph $\mathcal{G}_i$. 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 easily modeled by inserting dummy nodes between certain tasks and the source or the sink node respectively. These dummy nodes represent tasks with a certain execution time but which are not allocated to any PE. Thus, by meeting the global deadline, all the local deadlines and release times are guaranteed [17].

Regarding FPS tasks, we use the model from [7], which considers arbitrary deadlines and release times, and also takes into account dependencies. The tasks' priorities are specified by the designer. Thus, for each task τ_i we assume that we know its period T_i and deadline D_i.

III. System Model

We consider architectures composed of a set $\mathcal{N}$ of PEs which share a broadcast communication channel. Every PE $N_i \in \mathcal{N}$ consists, among others, of a communication controller and a CPU. We assume that the applications are separated using a temporal- and space-partitioning scheme similar to IMA. 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. A detailed discussion about space-partitioning is available in [3].

Each application $\mathcal{A}_j$ is allowed to execute only within its defined partition P_j. Although each SC application is running on a different partition to ensure separation, we assume that tasks from the NC applications can share the same partition. On a processing element N_i, a partition P_j is defined as the sequence 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 of application $\mathcal{A}_j$ mapped to N_i are allowed to use the PE. 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 node. Several MFs are combined together in a system cycle that is repeated periodically, with a period T_{cycle}.

In Fig. 1d we have 2 PEs, N_1 and N_2, and 3 applications, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$. Application $\mathcal{A}_2$ has 3 tasks, τ_5, τ_6 and τ_7. Task τ_6 executes in partition slices p_{12}^1 and p_{12}^2 allocated on N_1. When a task does not complete during the allocated slice, as it is the case with τ_6, its execution is suspended. The time overhead due to partition switching is denoted with t_O. Our approach takes into account the partition switching overheads. Tasks τ_5 and τ_7 execute in partition slices p_{22}^1 and p_{22}^2, respectively, allocated on N_2. 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. In the example in Fig. 1d all application tasks execute within one MF. However, an application can extend its execution over several MFs, as long as the deadlines are satisfied.

Each partition can use its own scheduling policy. We assume that SC applications are scheduled using non-preemptive static-cyclic scheduling, whereas NC applications are scheduled using preemptive fixed-priority scheduling. The start time of each SC task τ_i in the schedule table is denoted by t_i. The NC tasks have unique priorities denoted by $prio_i$.

We consider that processing elements are interconnected using a broadcast bus. We assume that the communication protocol has mechanisms to enforce partitioning at the bus level. For example, space partitioning is attained in SAFEBus [18] by mapping the messages to unique locations in the inter-module memory, protected by a memory-mapping hardware in the host, while temporal partitioning is achieved in TTP [1] by enforcing a Time-Division Multiple Access scheme.

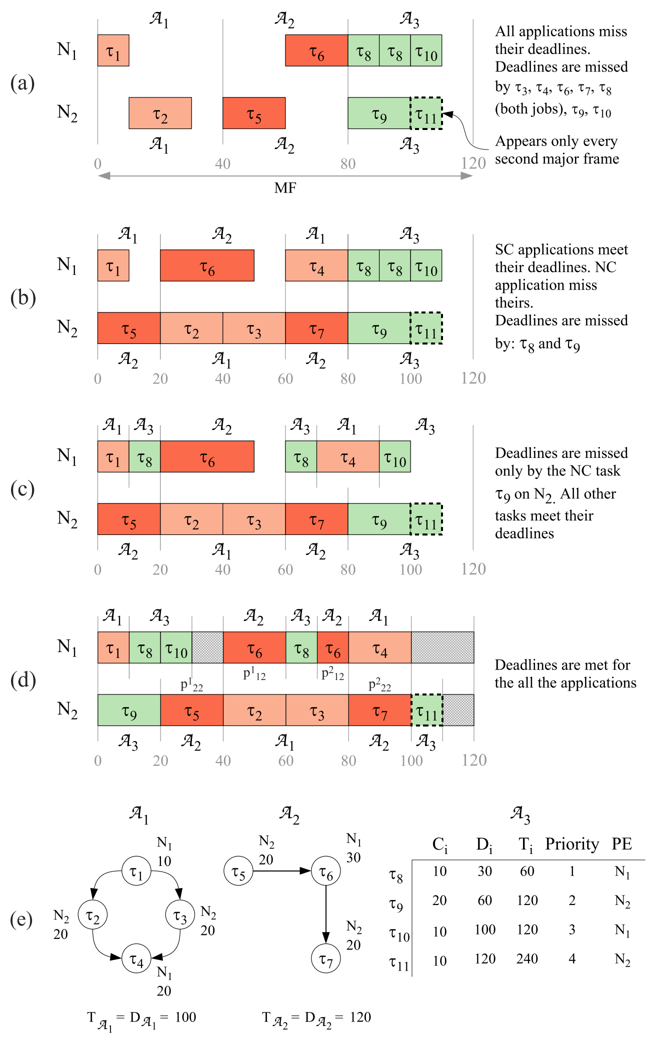
We have shown how realistic bus protocols such as TTP [17] and FlexRay (that combines TT and ET traffic) [14] can be taken into account during the design. However, in this paper we assume a more general bus access scheme, called *Universal Communication Model* (UCM) [6], where the communication cycle is partitioned into static and dynamic phases, see [7] for more details. The SC applications will transmit messages in the static phases, whereas the NC applications use the dynamic phases.

IV. Problem Formulation

The problem we are addressing in this paper can be formulated as follows: given (1) a set Γ of applications of mixed criticality levels, (2) an architecture consisting of a set $\mathcal{N}$ of processing elements, (3) the size of the major frame T_{MF} and (4) the application cycle T_{cycle}, we are interested to find an implementation Ψ such that all applications meet their deadlines. Deriving an implementation Ψ means deciding on (1) the set $\mathcal{P}$ of partition slices on each processor and (2) the set of schedules $\mathcal{S}$ for the SC applications.

Let us illustrate the problem using the example in Fig. 1 where we have two SC applications, $\mathcal{A}_1$ and $\mathcal{A}_2$, and one NC application, $\mathcal{A}_3$ (see Fig. 1e). For the SC applications, each task has next to it the PE it is mapped to and the worst-case execution time. The period and deadline for the applications are presented under the application graph. The NC tasks are scheduled using FPS and thus have their worst-case execution time C_i, deadline D_i, period T_i, priority and their PE, specified in the table. We have set T_{MF}=T_{cycle}=120 ms. To simplify the discussion, we assume that all NC tasks are released at time 0, that there is no partition switch overhead and the communication costs are ignored.

Note that, we consider the mapping of tasks to processing elements as fixed, and given as indicated in the figure. Very often, the mapping decision is taken based on the experience and preferences of the designer, considering aspects such as the functionality implemented by the task and the type of processing elements available, legacy constraints, proximity to sensors and actuators. This could be the reason, for example, that tasks τ_1 and τ_2 of $\mathcal{A}_1$ are mapped to different


(a)
N1
N2
All applications miss their deadlines. Deadlines are missed by τ3, τ4, τ6, τ7, τ8 (both jobs), τ9, τ10
Appears only every second major frame
MF
(b)
SC applications meet their deadlines. NC application miss theirs. Deadlines are missed by: τ8 and τ9
(c)
Deadlines are missed only by the NC task τ9 on N2. All other tasks meet their deadlines
(d)
Deadlines are met for the all the applications
(e)
T_A1 = D_A1 = 100
T_A2 = D_A2 = 120

	C_i	D_i	T_i	Priority	PE
τ_8	10	30	60	1	N_1
τ_9	20	60	120	2	N_2
τ_{10}	10	100	120	3	N_1
τ_{11}	10	120	240	4	N_2

Figure 1: Motivational example

processing elements. Many tasks, however, do not exhibit certain particular features or requirements which lead to an obvious mapping decision. We plan to address the problem of mapping optimization in our future research.

A simple way to do the partitioning is to divide the major frame equally among the 3 applications and to use the same partitioning slices on each PE, as depicted in Fig. 1a. The thin light grey lines are the borders for the partitions slices. Above, respectively under, each partition slice is specified the application it is assigned to. In this case, none of the applications meet their deadlines. Tasks τ_3 and τ_4 from $\mathcal{A}_1$ and τ_7 from $\mathcal{A}_2$ do not even fit into the system cycle. Note that the deadlines for the NC tasks are measured from their release time. Task τ_8 is released twice during the MF, at time 0 and 60. Task τ_{11} is released every second MF. The scheduling of SC tasks and the schedulability analysis for the NC tasks are presented in the next section.

In order to better accommodate the SC applications, we can try to adjust the size of the slices and introduce a new

```
TPO(Γ, $\mathcal{N}$)
1 $\mathcal{P}^\circ$ = InitialSolution(Γ, $\mathcal{N}$)
2 $\mathcal{P}$ = SimulatedAnnealing(Γ, $\mathcal{N}$, $\mathcal{P}^\circ$)
3 $\mathcal{S}$ = ListScheduling(Γ, $\mathcal{N}$, $\mathcal{P}$)
4 return  $\Psi = <\mathcal{P}, \mathcal{S}>$
```

Figure 2: Optimization strategy

slice for $\mathcal{A}_1$ on N_1 and for $\mathcal{A}_2$ on N_2, as shown in Fig. 1b. The SC applications meet their deadlines, but the NC tasks τ_8 and τ_9 miss in this case theirs. Note that although the slices have the same sizes on the two PEs, they are assigned to different applications.

Fig. 1c presents a way to make the NC task τ_8 on N_1 meet its deadline. The extra space from the first partition slice associated to $\mathcal{A}_1$ on N_1 is assigned to $\mathcal{A}_3$, and the second partition slice of $\mathcal{A}_1$ is shifted to the right. In this case, both jobs of τ_8 will meet their deadlines. However the NC task τ_9 on N_2 still has a deadline miss.

With the solution proposed in Fig. 1d the partitioning has been optimized such that all deadlines are met. In addition, we have also created 3 unused partition slices, depicted with a light grey rectangle, which can be used, for example, for future upgrades. This was managed by moving the time partition slice for the NC task τ_9 at the beginning of the partition table on PE N_2 and by splitting the SC task τ_6 on PE N_1 to execute in two different partition slices.

This example shows that the sequence and length of the partition slices has to be carefully optimized in order to find schedulable implementations.

V. Optimization of Time-Partitions

The problem presented in the previous section is NP-complete [19]. Its complexity depends not only on the number of tasks and processors, but also on the number of partition slices on each processor. In order to solve this problem, we will use the optimization strategy Time-Partitioning Optimization (TPO) from Fig. 2. TPO takes as input a set of applications Γ and the set of processing elements $\mathcal{N}$, and returns the implementation Ψ consisting of the set of partitions slices $\mathcal{P}$ on each processor and the schedules $\mathcal{S}$ for the SC applications. Our strategy has 3 steps:

(1) in the first step, we determine an initial set of partition slices $\mathcal{P}^\circ$, line 1 in Fig. 2. $\mathcal{P}^\circ$ consists of a simple straight forward 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 partitions P_{ij} thus allocated have the same length and they are distributed with a period equal to the smallest period of a task from $\mathcal{A}_j$ mapped to N_i.

(2) In the second step, we use a Simulated Annealing meta-heuristic to determine the set of partition slices $\mathcal{P}$ such that the applications are schedulable and the unused partition space (potentially used for future upgrades) is maximized (Section V-A). The alternatives provided by Simulated Annealing are evaluated using RTA (Section V-C) and List Scheduling (Section V-B).

(3) Finally, given the optimized partitions $\mathcal{P}$ obtained in line 2 in Fig. 2, we use a List Scheduling heuristic (presented in Section V-B) to determine the schedule tables for the SC applications.

A. Simulated Annealing

Simulated Annealing (SA) is an optimization meta-heuristic that tries to minimize the cost function in order to find the global optimum by randomly selecting neighboring solutions of the current solution [20]. The algorithm presented in Fig. 3 takes as input the set of application Γ, the architecture $\mathcal{N}$ and the initial partitioning $\mathcal{P}^\circ$, and returns the best solution found $\mathcal{P}^{best}$, i.e., with the smallest cost function (see line 8 in Fig. 3). In order to escape local minima, worse solutions are sometimes accepted with a probability depending on a control parameter called temperature and on the deterioration of the cost function (see lines 10 to 13 in Fig. 3). Before we call SA we merge all NC tasks into a single application, since the NC tasks are allowed to share a partition.

The algorithm is inspired by the process of annealing metals, a thermal process in which a metal is heated past its melting point and then carefully cooled down so that the particles arrange themselves with lower internal energy than the initial solution [20]. The cooling rate of the process influences the quality of the result. The cooling schedule of SA is defined by the initial temperature TI, the temperature length TL, the cooling ratio ε and the stopping criterion. The temperature length TL and the cooling ratio ε decide how fast will the temperature drop. We use a time limit as a stopping criterion (line 17).

We have defined our cost function as follows:

$$CostF(\mathcal{P}) = w_{SC} \times \delta_{SC} + w_{NC} \times \delta_{NC} \quad (1)$$

where δ_{SC} is the degree of schedulability for SC applications and δ_{NC} is the degree of schedulability for NC applications. These are summed together into a single value using the weights w_{SC} and w_{NC}. In case an application is not schedulable, its corresponding weight is a very big number, i.e., a "penalty" value. This allows us to explore unfeasible solutions (which correspond to unschedulable applications) in the hope of driving the search towards a feasible region. In case an application $\mathcal{A}_i$ is schedulable, we use for the weight a value given by the designer, depending on the importance of the application. For example, in our experiments we have used weights for the SC application which are several times greater than those for the NC applications. The degree of schedulability is calculated as:

$$\delta_{SC/NC} = \begin{cases} c_1 = \sum_i \max(0, R_i - D_i) & if c_1 > 0 \\ c_2 = \sum_i (R_i - D_i) & if c_1 = 0 \end{cases} \quad (2)$$

SimulatedAnnealing($\Gamma, \mathcal{N}, \mathcal{P}^\circ$)

```
temperature = initial temperature $TI$
$\mathcal{P}^{now}$ = $\mathcal{P}^{best}$ = $\mathcal{P}^\circ$
repeat
  for $i = 1$ to temperature length $TL$ do
    generate a random neighbor solution $\mathcal{P}'$ of $\mathcal{P}^{now}$
    $delta$ = $CostF(\mathcal{P}')$ - $CostF(\mathcal{P}^{now})$
    if delta $< 0$ then
      $\mathcal{P}^{now}$ = $\mathcal{P}^{best}$ = $\mathcal{P}'$
    else
      generate $q$ = random (0, 1)
      if $q < e^{-delta/temperature}$ then
        $\mathcal{P}^{now}$ = $\mathcal{P}'$
      end if
    end if
  end for
  $temperature$ = $\varepsilon \times temperature$
until stopping criterion is met
return $\mathcal{P}^{best}$
```

Figure 3: The Simulated Annealing algorithm

For SC applications δ_{SC} is computed at application level, and thus R_i is the response time of the application (i.e., the finishing time of the sink node) as resulted from List Scheduling (see Section V-B), while D_i is the deadline of the application. δ_{NC} is computed at task level. Thus, R_i is the worst-case response time and D_i is the deadline of each task. The response time for each task is computed according to the response time analysis presented in Section V-C.

If at least one NC task (or SC 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 tasks (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 neighboring solutions of the current solution $\mathcal{P}^{now}$ are generated using design transformations (or "moves") applied to $\mathcal{P}^{now}$. There are 4 types of moves: *resize*, *swap*, *join* and *split*. The moves are applied to a randomly selected partition slice from a randomly chosen PE.

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

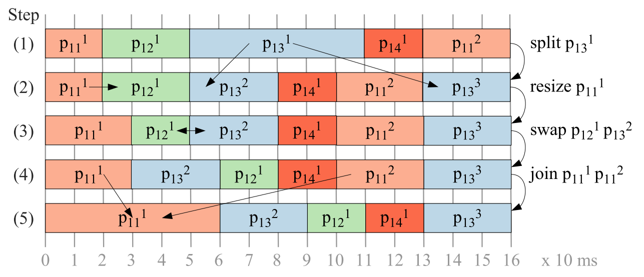


Figure 4: Move examples

slice to the end of the MF. Together with the 4 types of basic moves, we also apply "improved moves". An "improved move" is intended to accelerate the search by performing several basic moves at once.

Fig. 4 depicts the basic moves as they are sequentially performed on a single PE, namely N_1. As mentioned, the notation p_{ij}^k means the kth 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_{11}^1 and p_{11}^2. The current solution $\mathcal{P}^{now}$ is presented in Step 1 in Fig. 4. The first move is the *split* move, which is performed on the partition slice p_{13}^1 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_{11}^1 at the expense of p_{12}^1. The third move is a *swap* of slices p_{12}^1 and p_{13}^2. The result is shown in the 4th 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_{11}^1 and p_{11}^2 slices. Once a move has been performed on the partition set $\mathcal{P}^{now}$, the resulted partition set $\mathcal{P}'$ is evaluated using the cost function from (1), which is executed using List Scheduling and Response Time Analysis, presented in the next two sections, respectively.

B. List Scheduling algorithm

SC applications are scheduled using static-cycling non-preemptive scheduling. Given a partition set $\mathcal{P}$, we use a List Scheduling (LS)-based heuristic to determine the schedule tables $\mathcal{S}$ for each SC 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 [17] to sort L_{ready}.

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. Thus, LS is applied to each SC application. We have modified the classical LS algorithm to take into account the time partitions. Thus, each application is scheduled separately and its tasks are allowed to use only the corresponding partitions slices from $\mathcal{P}$. If a partition slice finishes before a SC task has completed its execution (as is the case with $\tau_6 \in \mathcal{A}_2$ in Fig. 1d), we assume that the task is suspended and will continue its execution in the next partition slice allocated to its application. 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 static segment of the UCM bus.

C. Response Time Analysis

NC applications are scheduled using FPS. To determine the schedulability of NC applications we use a response-time analysis [21] to calculate the worst-case response time R_i of every NC task τ_i, which is compared to its deadline D_i. The basic analysis presented in [21] has been extended over the years. For example, the state-of-the-art analysis from [22] considers arbitrary arrival times and deadlines, offsets and synchronous inter-task communication (where a receiving task has to wait for the input of the sender task). Audsley and Wellings [23] have proposed a schedulability analysis for FPS tasks using temporal partitioning (IMA), which, when analyzing a FPS task in a certain partition, considers the other time-partitions as higher priority tasks. This analysis assumes that the deadlines are smaller or equal to the periods, that the tasks are independent, and that the start times of partition slices within a major frame are periodic. Pop et al. [7] have proposed a schedulability analysis for ET tasks, which extends the schedulability analysis with static and dynamic offsets in [22] to consider the influence of the TT tasks on the worst-case response times of the ET tasks.

In this paper, we have extended the analysis from [7] to consider the influence of time-partitions on the schedulability of the NC tasks. In [7], the authors introduce the notion of *ET demand* and *ET availability*, used to compute the length of the busy window w_i, which is needed in order to compute the worst-case response time R_i of a task τ_i. The busy window w_i is the longest time interval during which tasks of priority equal or greater than τ_i are continuously executing [24]. The worst-case response time is determined using (3), considering a certain length of the busy window w_i, and all the higher priority tasks:

$$R_i = w_i - \varphi_i - (p-1) \times T_i + \phi_i \tag{3}$$

where p is the number of activations of task τ_i in the busy window w_i, T_i is the period of τ_i, the offset ϕ_i is the earliest activation of τ_i relative to the occurence of the triggering event and the phase φ_i is the time interval between the critical instant and earliest time of the first activation of τ_i. The worst-case response time R_i for the task τ_i is the maximum value of the result in (3), considering all the critical instants initiated by higher priority tasks and by τ_i and also all the job instances. During the calculation, if the *available* time does not satisfy the *demand* of τ_i then the algorithm increases iteratively the length of the busy window w_i which is analyzed [7].

Similar to the notion of *ET demand* from [7], we introduce *NC demand*, associated with a NC task τ_i on a time interval t, as the maximum amount of CPU time which can be *demanded* by higher or equal priority NC tasks and by τ_i. Fig. 5 shows the analysis for a task τ_i, considering the busy window that starts at the critical instant $qT_i + t_c$, initiated by task τ_a and ends at the moment $qT_i + t_c + w_i$, when all the higher priority tasks (τ_a and τ_b) and τ_i itself have finished execution, and when all the partition slices interrupting τ_i have finished.

During the busy window w_i, the *demand* H_i associated with task τ_i scheduled in a partition P_k is equal with the length of the busy window which would result when considering that P_k would be the only partition on the processor. Thus, similar to [7] and [22], the *NC demand* is:

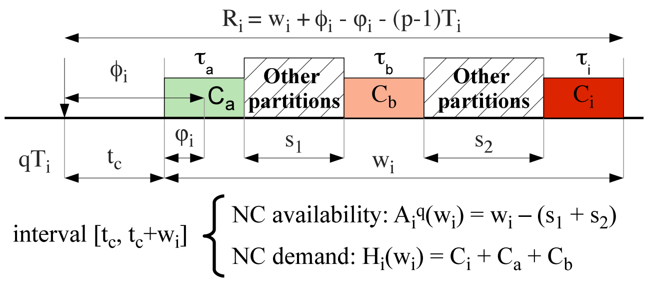
$$\begin{aligned} H_i(w_i) = B_i + (p - p_{0,i} + 1) \times C_i + \\ W_i(\tau_i, w_i) + \sum_{\forall (a \in \mathcal{A}_a \neq \mathcal{A}_i)} W_a^*(\tau_i, w_i) \end{aligned} \tag{4}$$

where B_i is the maximum blocking time for τ_i. The job activations of task τ_i during w_i are denoted with p and positive values are assigned to instances arriving after t_c, while zero and negative values indicate the instance arrived before t_c. Thus, $p_{0,i}$ is the index of the first pending instance of τ_i and is computed as follows:

$$p_{0,i} = 1 - n_{ia} = 1 - \left\lfloor \frac{J_i + \varphi_i}{T_i} \right\rfloor, \tag{5}$$

where n_{ia} is the number of pending τ_i jobs at t_c, during the busy window w_i initiated by τ_a. $W_i(\tau_i, w_i)$ is the interference from higher priority tasks hp(τ_i) in the same application $\mathcal{A}_i$ as τ_i:

$$W_i(\tau_i, w_i) = \sum_{j \in hp(\tau_i)} \left(\left\lfloor \frac{J_j + \varphi_i}{T_i} \right\rfloor + \left\lceil \frac{w_i - \varphi_i}{T_i} \right\rceil \right) \times C_j \tag{6}$$



R_i = w_i + ϕ_i - φ_i - (p-1)T_i
ϕ_i
τ_a
C_a
Other partitions
τ_b
C_b
Other partitions
τ_i
C_i
qT_i
t_c
φ_i
s_1
w_i
s_2
interval [t_c, t_c+w_i]
NC availability: A_i^q(w_i) = w_i – (s_1 + s_2)
NC demand: H_i(w_i) = C_i + C_a + C_b

Figure 5: Availability and demand

and $W_a^*(\tau_i, w_i)$ is the worst-case interference from higher priority $hp_a(\tau_i)$ tasks from other applications than the application $\mathcal{A}_i$ that τ_i belongs to:

$$W_a^*(\tau_i, w_i) = max(W_k(\tau_i, w_i)), \forall k \in hp_a(\tau_i) \tag{7}$$

Fig. 5 shows that the *NC demand* of task τ_i during the busy window w_i is the sum of the worst case execution times of the higher priority NC tasks, C_a and C_b, and the worst-case execution time C_i of τ_i.

We extend the concept of *availability* from [7] as the processing time available during w_i for P_k. Considering we are using time partitions and that task τ_i can execute only during its own partition P_k, the *availability* is computed by subtracting from w_i the time reserved for the "other partitions". In Fig. 5 the other partitions are illustrated with hashed rectangles and their duration denoted with s_1 and s_2.

The response time analysis also determines the worst-case response time R_m of each message m transmitted on the dynamic segment of the UCM bus. For details, see [7].

VI. Experimental Evaluation

For the evaluation of our proposed algorithm we used 10 synthetic benchmarks and 2 real life case studies. The Time-Partition Optimization (TPO) 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 TPO strategy in terms of its ability to find schedulable implementations. Thus, we have used 5 synthetic benchmarks with 3 to 5 SC applications (with a total of 15 to 53 SC tasks). All the NC tasks have been merged into a single NC application, with 5 to 9 tasks. The resulted mixed-criticality system has been mapped on architectures ranging from 2 to 6 processing elements. The mapping has been done such that the utilization on the PEs is balanced and the communication over the bus is minimized. The execution times and message lengths were assigned randomly within the 1 to 19 ms and 1 to 5 bytes ranges, respectively. The weights used for computing the cost function were $w_{SC} = 400$ for SC applications and $w_{NC} = 100$ for NC tasks (see Section V-A).

We have used two time limits for the experiments: 10 minutes and 120 minutes. The results obtained with TPO using a time limit of 120 minutes are presented in Table I, under the heading "TPO, 120 min. time limit".

We were interested to determine the quality of our Simulated Annealing-based (SA) TPO strategy. Hence, we have used an exhaustive search to determine the optimal solutions. Since the runtime of the exhaustive search is prohibitively large, we were only able to run it for smaller examples, lines 1, 6 and 7 in Table I. In these cases, our SA-based approach is capable of obtaining (in 120 minutes) solutions which are very close to the optimum. For the benchmarks in lines 1, 6 and 7 the difference in term of the cost function is only 4.51%, 0.16% and 1.9%, respectively.

Together with TPO, Table I also presents the results obtained using a Straightforward Solution (SS), which implements the approach from the InitialSolution function presented in Section V. SS is an approach that a good designer would use if TPO would not be available. Columns 2 and 4 in Table I present the number of SC applications, and the NC tasks, respectively. The number of schedulable applications and tasks (out of the total) obtained by our proposed TPO strategy are presented in columns 8 and 9, respectively, while columns 6 and 7 present the results obtained using SS. Columns 10 and 11 represent the percentage increase in the degree of schedulability for SC applications, Δ_{SC}, and NC tasks, Δ_{NC}, (see Section V-A) as obtained by the TPO strategy compared to the SS, considering a time limit of 120 minutes. A negative value for Δ_{NC} means that our optimization has decreased the degree of schedulability for the NC tasks in order to guarantee that all SC applications are schedulable. Note that the NC tasks are still schedulable in this case, but their response times have increased, compared to SS, which over-dimensioned the NC partitions. Column 12 represents the average of the percentage increase in the degree of schedulability for the whole system.

We have also run TPO with a time limit of 10 min. TPO is able to obtain schedulable solutions in all cases, except for the case study in line 4 in Table I. The average deviation of the percentage increase of the cost function (as captured by (1)) for the schedulable results, compared to the results obtained with TPO using a 120 min. time limit, is of 10.48%.

As we can see from "Set 1", SS which does not perform optimization, is not able to find schedulable implementations. For example, for the largest benchmark, with 5 SC application and 9 NC tasks mapped on 6 PEs only 3 out of 5 SC applications are schedulable. All the NC tasks are schedulable. Note that SS leads to schedulable NC implementations. This is because it distributes the partition slices to match the smallest period of the tasks. However, since the slices have equal lengths, there is a lot of wasted space in the schedules of SC applications, which leads to missed SC deadlines. However, by applying our proposed TPO approach, we are able to optimize the time partitions such that all applications are schedulable. We have measured the ability of TPO to improve over SS by using a percentage average increase in the degree of schedulability over all applications, presented in the last column. As we can see there is a dramatic increase in the degree of schedulability over all applications, when using TPO. This means that we can potentially implement the applications on a slower (cheaper) architecture.

In the second set of applications, labeled "Set 2", we were interested to see how TPO performs as the utilization of the system increases. We have mapped 2 to 6 applications on the same architecture of 4 PEs. As we can see, TPO is able

Table I: Experimental results for benchmarks

Set	SC Apps	SC Tasks	NC Tasks	PEs	SS Sched. SC Apps	SS Sched. NC Tasks	TPO, 120 min. time limit Sched. SC Apps	TPO Sched. NC Tasks	TPO Δ_{SC}	TPO Δ_{NC}	TPO avg. % increase in δ
1	3	15	5	2	1 of 3	All	All	All	1709.76	-44.00	832.88
	3	20	6	3	1 of 3	All	All	All	107.94	-53.23	27.36
	4	34	6	4	None	All	All	All	169.68	7.14	88.41
	4	40	10	5	None	All	All	All	147.54	-0.40	73.57
	5	53	9	6	3 of 5	All	All	All	542.78	14.66	278.72
2	1	6	6	4	All	All	All	All	78.38	0.00	39.19
	2	12	6	4	All	All	All	All	59.20	-2.87	28.17
	3	20	6	4	None	5 of 6	All	All	518.06	1453.85	985.96
	4	30	6	4	1 of 4	All	All	All	211.66	0.00	105.83
	5	34	6	4	2 of 5	5 of 6	All	All	466.36	673.33	569.85
3	3	19	5	3	None	All	All	All	227.33	0.57	113.95
	4	19	6	3	All	All	All	All	135.29	-11.56	61.87

to find schedulable implementations even as the utilization increases.

Finally, we have also used 2 real life benchmarks derived from the Embedded Systems Synthesis Benchmarks Suite (E3S) version 0.9 [25]. We have used the *telecom-mocsyn* and *auto-indust-cowls* benchmarks. In the case of *telecom-mocsyn* test case, the applications numbered as 0, 1, 2 and 4 were used as SC, and applications numbered as 3, 5, 6 and 7 were merged into one NC application. In the case of the *auto-indust-cowls*, the first 3 applications are considered SC, while the last one is NC. In both cases the applications are mapped on an architecture of 3 PEs. The results obtained from these real-life benchmarks confirm the results of the synthetic benchmarks.

VII. Conclusions and Future Work

In this paper we have presented a Simulated Annealing based algorithm for the optimization of time-partitions for mixed-criticality real-time distributed embedded systems. The algorithm considers that the applications are separated using a temporal- and spatial-partitioning scheme similar to IMA, that each application runs in a separate time partition, that the SC applications are scheduled using static-cycling scheduling, while the NC applications are scheduled using fixed-priority preemptive scheduling. Two real life examples have been used, as well as 10 synthetic benchmarks to show the effectiveness of the proposed algorithm. We have shown that only by optimizing the sequence and length of the time partitions we are able to obtain schedulable implementations.

The most important extension in our future work will be to consider certification costs. We will extend existing cost models from literature [26] to take into account the additional effort required for certification at a given Safety-Integrity Level. If two applications of different SILs share a partition, they will have to be certified at the highest SIL level among the two, increasing thus the overall development costs. Our future approach will allow the sharing of partitions, which could lead to reduced system costs, and will consider the increased development costs due to increased certification efforts. In this paper we have assumed that the mapping is given, but we plan to include the problem of assigning tasks to processing elements in the overall partitioning optimization problem. Also, in our future work we will investigate which is the best optimization approach, for example using Tabu Search instead of Simulated Annealing.

References

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

[2] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, P. Stanfill, D. Stuart, and R. Urzi, "A research agenda for mixed-criticality systems," in *Cyber-Physical Systems Week*, April 2009.

[3] J. Rushby, "Partitioning for avionics architectures: Requirements, mechanisms, and assurance," NASA Langley Research Center, NASA Contractor Report CR-1999-209347, Jun. 1999, also to be issued by the FAA.

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

[5] N. Audsley, K. Tindell, and A. Burns, "The end of the line for static cyclic scheduling," in *Proc. of Euromicro Workshop on Real-Time Systems*, 1993, pp. 36–41.

[6] T. Demmeler and P. Giusto, "A universal communication model for an automotive system integration platform," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*. IEEE, 2002, pp. 47–54.

[7] T. Pop, P. Pop, P. Eles, and Z. Peng, "Analysis and optimisation of hierarchically scheduled multiprocessor embedded systems," *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 37–67, 2008.

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

[9] 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*, vol. 0, pp. 89–98, 2010.

[10] 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*, vol. 17, no. 3, pp. 389 –402, march 2009.

[11] 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*, 2000, pp. 14 –23.

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

[13] 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*, 1999, pp. 178–182.

[14] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei, “Timing analysis of the FlexRay communication protocol,” *Real-Time Systems*, vol. 39, no. 1-3, pp. 205–235, 2008.

[15] 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, pp. 13 –22.

[16] 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*, 2009, pp. 291–300.

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

[18] K. Hoyme and K. Driscoll, “SAFEbus,” *IEEE Aerospace Electronic Systems Magazine*, vol. 8, pp. 34–39, 1993.

[19] J. D. Ullman, “NP-complete scheduling problems,” *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, 1975.

[20] E. Burke and G. Kendall, *Search Methodologies*. Springer Science + Business Media, 2005, ch. 7.

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

[22] J. Palencia and M. Gonzalez Harbour, “Schedulability analysis for tasks with static and dynamic offsets,” in *Proc. of Real-Time Systems Symposium*, 1998, pp. 26 –37.

[23] N. Audsley and A. Wellings, “Analysing APEX applications,” in *Real-Time Systems Symp.*, 1996, pp. 39 –44.

[24] C. Fidge, “Real-time schedulability tests for preemptive multitasking,” *REAL-TIME SYSTEMS*, vol. 14, no. 1, pp. 61–93, JAN 1998.

[25] R. Dick, “Embedded system synthesis benchmarks suite,” http://ziyang.eecs.umich.edu/d̃ickrp/e3s/.

[26] J. Axelsson, “Cost models for electronic architecture trade studies,” in *Proc. of ICECCS*, 2000, pp. 229–239.