Architecture Exploration of Real-time Systems Based on Multi-Objective Optimization

Rahma Bouaziz*, Laurent Lemarchand†, Frank Singhoff†, Bechir Zalila*, Mohamed Jmaiel*‡
* ReDCAD Laboratory, University of Sfax, ENIS, B.P. 1173, 3038 Sfax, Tunisia
†Lab-STICC Laboratory, University of Bretagne Occidentale, UMR CNRS 6285, F-29200 Brest, France
‡Research Center for Computer Science Multimedia and Digital Data Processing of Sfax, B.P. 275, Sakiet Ezzit, 3021 Sfax, Tunisia
bouaziz.rahma@redcad.org, {laurent.lemarchand, singhoff}@univ-brest.fr, {bechir.zalila, jmaiel.mohamed}@enis.rnu.tn

***Abstract*—This article deals with real-time embedded system design and verification. Real-time embedded systems are frequently designed according to multi-tasking architectures that have timing constraints to meet. The design of real-time embedded systems expressed as a set of tasks raises a major challenge since designers have to decide how functions of the system must be assigned to tasks. Assigning each function to a different task will result in a high number of tasks, and then in higher preemption overhead. In contrast, mapping many functions on a limited number of tasks leads to a less flexible design which is more expensive to change when the functions of the system evolve. This article presents a method based on an optimization technique to investigate the assignment of functions to tasks. We propose a multi-objective evolution strategy formulation which both minimizes the number of preemptions and maximizes task laxities. Our method allows designers to explore the search space of all possible function to task assignments and to find good trade-offs between the two optimization objectives among schedulable solutions. After explaining our mapping approach, we present a set of experiments which demonstrates its effectiveness for different system sizes.**

***Keywords*—Real-time Embedded Systems, Architecture exploration, Multi-Objective Optimization, PAES, Scheduling Analysis**

I. Introduction

This article is about the design of real-time embedded critical systems. Despite significant advances in the development of such systems, designing them is still a challenging task since they must fulfill different constraints along with the achievement of their main mission. In some domains, real-time embedded critical systems may consist of a large number of time constrained functions [1]. During software architecture design, these functions must be mapped onto tasks that will run the functions on the top of a real-time operating system (RTOS). The objective of our work is to guide the designer by proposing solutions to map functions to tasks.

a) Problem Statement: Mapping the functional specifications to a software architecture manually may be time-consuming and error-prone regarding to the number of functions involved.

Moreover, it is a non-trivial task, due to the large number of mapping solutions ranging from single-task architectures to concurrent multi-tasked architectures.

Indeed, there are two extreme tasking implementation strategies [2]:
(i) In the single-task implementation solution, the whole functional specification is executed in the context of a single task. Although this solution is quite simple and well suited to highly memory-constrained applications, it leads to a less flexible design which is expensive to change when the functions of the system evolve.
(ii) With the one-to-one mapping of functions to tasks, each function is assigned to a single task. Obviously, each task inherits timing parameters of the corresponding function. This second approach could be inefficient [2] because it leads to a high number of tasks which involves a significant memory consumption (e.g. stack size for each task) in addition to the excessive scheduler overhead due to context switching or preemptions.

Thus, the designer needs a trade-off between those two extreme assignment solutions in order to balance schedulability with flexibility of the design and performance overhead.

For a given set of functions, the solution space size (*i.e.* the number of all possible assignments to task sets) is defined by the *Bell number* [3] which is exponential with regard to the number of functions. Considering that the number of functions is usually quite high [1] (in most application it is in the order of hundreds), the number of assignment possibilities to explore is too large. Therefore, an exhaustive and exact search among all possibilities is not practicable.

Furthermore, when performing functional-to-architectural mapping, reducing preemptions and maximizing the laxity of tasks are conflicting goals. As an example, for a set of 2 functions f_1 and f_2, implementing them as separate tasks maximizes the laxity (the amount of time between the end of the execution of a task and its deadline, it is a measure of the flexibility available for scheduling a task) but produces preemption overhead. Grouping them in a single task, thus without any preemption, could reduce the laxity, and worse, the task could also become unschedulable.

b) Contributions of this Article: In this article, we propose a method to drive the designer by producing a set of design solutions. The approach aims both at reducing the number of preemptions for minimizing timing overheads and at maximizing the laxity of tasks in order to improve the schedulability of the design model.

As we face two conflicting objectives, a multi-objective optimization technique is applied to solve the mapping problem. From a set of functions, the proposed technique investigates a set of solutions in terms of task sets by function clustering that preserves schedulability while optimizing our two objectives.

Rather than exhaustively searching among all possible architectures, we propose to use an evolutionary strategy in order to search the schedulable and optimal solutions in the mapping search space. The proposed technique is based on the *Pareto Archived Evolution Strategy (PAES)* [4] for the approximation of the *non-dominated* front and the Cheddar tool [5] for the metrics computation and the timing constraints verification.

In this article, we assume a uniprocessor system with a preemptive scheduler. As we target critical systems, we make scheduling analysis based on a fixed priority policy with the Rate Monotonic priority assignment (RM) [6].

The remainder of the article is organized as follows. Section II defines the system models, notations and introduces the multi-objective optimization (MOO). Section III presents our approach, corresponding to a formulation of the PAES MOO algorithm for the addressed problem as well as a definition of the function to task assignment rules. Section IV contains experimental results. Section V presents the related work and section VI concludes the article.

II. Background and Notation

This section presents the notations and models we assume. Then it gives the basic concepts of the Multi-objective Optimisation (MOO) and introduces the PAES algorithm as the MOO technique used in our approach.

A. Functional and Architectural Models

At the design stage of real-time embedded systems, we distinguish two specification levels namely the functional and the architectural specification.

The *functional specification* defines the functions of the system and their real-time characteristics. Formally speaking, the functional model used in our work is defined by a set of *functional blocks* or *functions* and denoted by $\Gamma = \{F_1, F_2, ..., F_n\}$. We consider *atomic* functions, i.e. a function presents the smallest indivisible code unit. We assume also that all functions are periodic and independent. A function F_i is characterized by three parameters $F_i = (\gamma_i, \zeta_i, \delta_i)$ where γ_i is the maximum computation time, the activation period denoted as ζ_i is a fixed delay between two release times and δ_i is the deadline defined by the time limit in which the function must complete its execution.

At the *architectural level*, the designer defines the concrete model on which the functional abstractions must be assigned to a set of real-time tasks that will be run on the top of an RTOS [2]. We assume that the considered architectural model is based on the well-known task model of Liu and Layland [7]. It consists of a set of k periodic and independent tasks that we denote by $S = \{\tau_1, \tau_2, ..., \tau_k\}$ running on a uniprocessor platform. A task is defined by three parameters τ_i=(C_i, T_i, D_i): its capacity or worst case execution time C_i, its activation period T_i and its deadline D_i. We assume that all tasks are synchronous (i.e. all tasks are released at the same time). The scheduling analysis is made with the Rate Monotonic priority assignment (RM). Task parameters are computed according to the parameters $(\gamma_j, \zeta_j, \delta_j)$ of all functional blocks run by the task (see section III-C).

The scheduling analysis is performed by a scheduling simulation on the *hyperperiod* which is the least common multiple of task periods. Indeed, Leung and Merrill [8] proved that the hyperperiod is the simulation duration required in order to study the schedulability of a Liu and Layland task model system, under a fixed priority and preemptive scheduling policy.

B. Multi-Objective Optimization

Multi-objective optimization (MOO) [9] techniques deal with optimization problems that involve more than one objective to be optimized simultaneously with respect to a set of constraints.

Candidate solutions for a multi-objective optimization problem are evaluated by their performance in more than one objective [10], [11]. Then, the outcome of running MOO is a single or a set of solutions, with each solution representing a trade-off between objectives.

MOO has been applied in many fields, including product and process design, economics and logistics where an attempt to improve one objective leads to the degradation of the other and then, when decisions need to be taken in the presence of trade-offs between objectives.

1) Dominance Concept: For a MOO problem, a single solution that simultaneously optimizes all objectives rarely exist. Instead, a set of *non-dominated* solutions called also *Pareto optimal* solutions are searched for. As depicted in Figure 1, these solutions constitute the *Pareto Front* which represents the best trade-off set for the considered objectives. Two candidate solutions can be compared according to the *Pareto Dominance* concept [12]: a candidate solution c_1 ***dominates*** another candidate solution c_2 *if and only if* (i) c_1 is strictly better than c_2 for at least one of considered objective and (ii) c_1 is not worse than c_2 for any of the objectives.

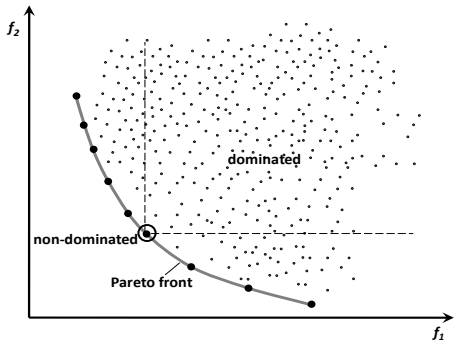

Fig. 1: Pareto front for two minimization objectives

Many algorithms have been developed to solve MOO problems [9], with a large amount of them derived from evolutionary algorithms (MOEA). The key points for those algorithms are their accuracy (how close to the Pareto Front

are the solutions they provide) and their efficiency (are the solutions numerous and well spaced over the whole Pareto front). As opposed to exact methods for MOO like multi-objective integer programming, they can be applied to problems with a very large search space

Evolutionary algorithms are metaheuristic optimization algorithms inspired from nature (swarm particle, ant colony, simulated annealing). As an example, genetic algorithms use biology-inspired mechanisms like *mutation* and *crossover* in order to refine a set of candidate solutions iteratively [13].

2) The PAES Algorithm: The Pareto Archived Evolution Strategy (PAES) [4] is a MOEA technique using archiving. It serves to find a set of solutions properly distributed over the whole spectrum of possible trade-offs between objectives and thus allows us to make architectural exploration.

It manipulates a single solution as opposed to other methods [9]. This is a key point in running time-consuming evaluation functions. The general PAES schema is outlined in the pseudo code of Algorithm 1. In many evolutionary

Algorithm 1: General form of PAES Algorithm

```
1  begin
2      Generate initial random solution c
3      Evaluate c and add it to the archive
4      repeat
5          Mutate c to produce new candidate solution m;
6          Evaluate m;
7          if (c dominates m) then
8              Discard m;
9          else if (m dominates c) then
10             Replace c with m; Add m to the archive;
11         else if (m is dominated by any member
12                 of the archive) then
13             Discard m;
14         else
15             Apply test (c,m,archive) to determine which
               becomes the new current solution and
               whether to add m to the archieve;
16         end if
17     until (termination condition is satisfied);
18 end
```

algorithms, a solution is represented by a set of parameters called *chromosome* (or *genotype*). The *encoding* of solutions (i.e. the data structure of a chromosome) is specific for each problem.

The PAES algorithm is based on a (1+1) evolution strategy which means that it maintains a single current solution (*parent*) and, at each iteration, generates a single new candidate (*offspring*) through a random mutation (line 5). This algorithm is confined to a local search i.e. it uses only a small change (mutation) operator that moves from a current solution to a nearby neighbour. The mutation procedure is also specific for each problem.

The candidate solutions are evaluated (line 6) according to the *fitness functions* and compared using the pareto dominance concept. A fitness function is used to indicate the quality of a solution according to an objective criteria. Each objective is defined by a fitness function. PAES maintains a list of some non-dominated solutions called *archive* used as reference set with respect to which each new candidate is being compared (lines 11,12). The algorithm iterates (lines 4/17) according to some convergence criteria of the fitness function values or based onto a fixed number of iterations.

III. Functional-To-Architectural Mapping Approach

A. Methodology Description

In this section, we present an overview of our methodology. As described in Figure 2, at the specification-level, (1) the designer provides the functional specification of the real-time system. This specification represents the input model of our approach. Then, (2) a first architectural model is proposed so that each functional block is assigned to only one software task. (3) a scheduling analysis with RM is performed on this initial architectural model by a scheduling simulation with the Cheddar tool. If this task set is schedulable then it will be considered as the initial current solution to the PAES algorithm. Otherwise, the designer must adjust the timing parameters of her/his functional model. (4) once the initial current solution is defined, we come to the architectural exploration and optimization step. This step involves the execution of the PAES algorithm, starting from the initial current solution to get at the end of the execution a set of different schedulable architectural models that constitute the Pareto front (5). From these solutions, the designer would choose the actual architecture.

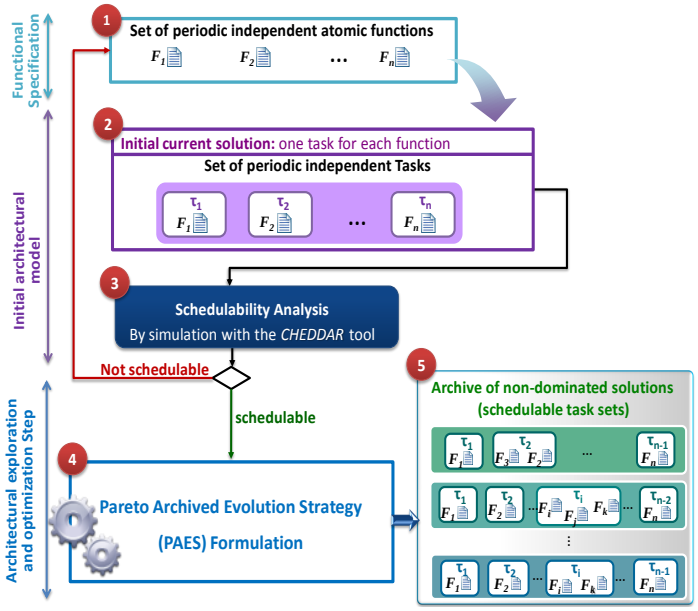

Fig. 2: Approach description

B. PAES Formulation of the Mapping Problem

As presented in Section II-B2, PAES relies on the definition of operators to solve a particular MOO problem. In this section, we specify those operators, *i.e* the fitness functions, the encoding of solutions, the initial current solution, and the mutation operator used to formulate the PAES for our problem.

1) Fitness Functions: According to our optimization goals, we consider two metrics: (1) the minimization of the number of preemptions to express the minimization of scheduling timing overheads and (2) the maximization of the tasks laxity to improve the design flexibility.

A preemption occurs when a higher priority task τ_i is released during the execution of a lower priority task τ_j and when the scheduler interrupts the execution of τ_j to allow the task τ_i to run [14]. A preemption causes a context switching which requires to run several processor instructions and then an important overhead to the overall task execution time. The fitness function relative to this metric enumerates the total number of preemptions in the scheduling simulation performed on the hyperperiod of the initial architectural model. We note this fitness function as follows:

$$f_1 = Min(number\_of\_preemptions)$$

The laxity indicates the maximum delay that can be taken by a task without exceeding its deadline [15]. For a task τ_i, it is defined as the difference between its deadline D_i and its worst-case response time R_i, or $D_i - R_i$. The worst-case response time [16] R_i is the maximum delay between the release time and the completion time of the task. The fitness function that represents this second metric computes the sum of laxity over all resulting tasks:

$$f_2 = Max(\sum_{i=1}^{k}(D_i - R_i)) \quad (k: \ number\ of\ tasks)$$

The PAES algorithm was designed to deal with only two kinds of problems, either maximization problems (i.e. all objectives are to be maximized) or minimization problems (i.e. all objectives are to be minimized). However, we have the first objective for minimization and the second one for maximization. To tackle this problem, we express the second objective as a minimization. As shown in the following equation, we have chosen the hyperperiod of the initial architectural model as a big constant from which the original objective will be subtracted. With such a method the minimization of the new expression would result in the maximization of the second objective:

$$f2 = Min(hyperperiod - \sum_{i=1}^{k}(D_i - R_i))$$

Fitness functions are computed using the Cheddar tool. Indeed, this tool provides the results of the scheduling simulation in terms of schedulability, the number of preemptions, the number of context switch and the worst-case response time of tasks.

2) Encoding: In our work, a solution is represented by means of *Integer Encoding* schema that is adapted in various previous work [12]. With such encoding, a chromosome is defined as an integer vector with n positions. n is the number of all functions. A chromosome represents the assignment of functions to tasks. Each position in the vector, called *gene*, corresponds to a particular function, i.e the i^{th} position represents the i^{th} function. The value held by a gene represents the index of the task to which the corresponding function belongs to. Thus, in our case, a chromosome represents a solution formed by k tasks ($k \leq n$), where each gene has a value in $\{1, 2, ..., k\}$.

Figure 3 shows an example of a chromosome that corresponds to a particular solution S. The chromosome size indicates the number of functions. In this example, the functional model is composed of 8 functions. Gene F_1 holds a value equal to 2, which means that the function F_1 is assigned to the task of index 2 (τ_2). The solution corresponding to this chromosome assigns F_4 to τ_1; F_1,F_3 and F_7 to τ_2; F_2 and F_6 to τ_3; F_5 and F_8 to τ_4.

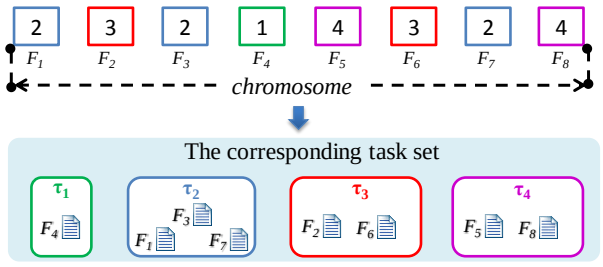

Fig. 3: Chromosome representation of a particular mapping solution S

This encoding is quite simple but it is redundant, i.e., a solution can be represented by different chromosomes. For example the same solution S encoded by the chromosome depicted in Figure 3 can be represented by other chromosomes, namely [1 2 1 3 4 2 1 4], [3 1 3 2 4 1 3 4], [2 4 2 3 1 4 2 1], etc. In other words, a solution may be represented by many chromosomes that maintain the same partition of functions independently of the indexes of tasks. To reduce all of the equivalent partitions to the same chromosome representation, we *normalize* the chromosome representation. This is made as follows: the task index of the function F_1 is always 1 (then all functions that belong to the same task of F_1 have as task index 1), then the task index of F_2 is 2, except if F_2 is in the same task with F_1, and so on. Figure 4 shows the normalized chromosome representation of the same task set mapping solution S.

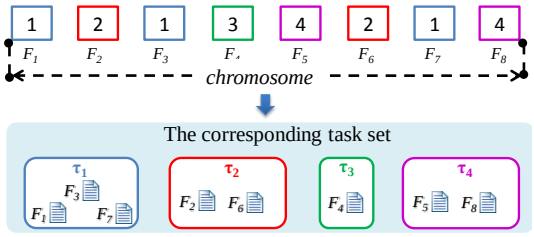

Fig. 4: The normalized chromosome representation of the same mapping solution S

The normalization procedure helps to compare two solutions without ambiguity. It must be applied on each new chromosome generated by mutation.

3) The initial current solution: consists in assigning each function to a different task, i.e., the number of tasks is equal

to the number of functions:

$$chrom = [1\ 2\ 3\ 4\ ..\ n]\ (n:\ number\ of\ functions).$$

4) Mutation Operator: We have adapted a random mutation operator that chooses a random position in the chromosome and changes the value of the corresponding gene in the selected point to a new random value. Algorithm 2 describes the mutation procedure proposed for our mapping problem. This procedure is implemented in order to generate only feasible solutions that satisfy the assignment constraints (given in Section III-C) and the schedulability of the resulting task set.

C. Function-to-Task Assignment Rules

In this section, we define the rules to assign functions to tasks. It consists in determining the task real-time scheduling parameters according to the timing parameters of the functions.

1) Initialization step: At the initialization step, each function is assigned to a task that will take the same parameters of the corresponding function.

$$\forall i \in \{1..n\} \quad F_i = (\gamma_i, \zeta_i, \delta_i)$$

$$\tau_i = (C_i, T_i, D_i) \quad \begin{cases} C_i = \gamma_i \\ T_i = \zeta_i \\ D_i = \delta_i \end{cases}$$

2) Generation of New Candidate Solutions: The mutation procedure allows us to explore different software architectures (task sets) through the assignment of functions to tasks. As described in the mutation algorithm 2, we cluster/separate functions to get a new configuration of task set. However, the mapping of the functions and the resulting task set parameters must preserve :

(i) The functional specification in terms of periodic activations of the functions.
(ii) The schedulability of the system (i.e the schedulability of the resulting task set)

In the sequel, we explain and motivate our mapping method and how we compute from the function parameters the parameters of each task, while taking into account the constraints outlined above.

Let consider a task $\tau_i = (C_i, T_i, D_i)$ of the initial model containing a single function $F_i = (\gamma_i, \zeta_i, \delta_i)$. Let see also a classic implementation of a periodic task with Ada [17]. The Listing 1 presents such an implementation for τ_i. The task is periodically released thanks to the DELAY UNTIL statement and then, calls the F_i procedure to run the function implemented by the task.

At a mutation action, another function $F_j = (\gamma_j, \zeta_j, \delta_j)$ is chosen to be assigned to the task τ_i. Let see how to set τ_i parameters.

Algorithm 2: Mutation algorithm for the function-to-task assignment approach

```
/* Mutation Algorithm: generate a
mutated solution from a current
solution                                   */
/* current Solution: n functions
mapped into k tasks (n ≥ k)                */
1  begin
2      Choose randomly a function F_i (1 ≤ i ≤ n);
       /* we have chrom[F_i] = τ_j i.e. the
       function F_i is assigned to the task
       τ_j */
3      Determine the set of harmonic tasks with the chosen
       function F_i called harmonic_task_set;
4      if (harmonic_task_set = {τ_j}) then
           /* i.e. the function F_i is not
           harmonic with any task, only the
           task to which it belongs        */
5          Restart the algorithm with another function
           chosen randomly;
6      else
7          Choose randomly a task τ_m over tasks in the set
           harmonic_task_set (including τ_j = chrom[F_i]);
8          if (τ_m ≠ τ_j) then
9              chrom[F_i] ← τ_m; /* F_i is moved to
               the task τ_m                */
10         else if (The function F_i is not alone in τ_j) then
11             Create a new task τ_{k+1};
12             chrom[F_i] ← τ_{k+1}; /* F_i is isolated
               in the new task τ_{k+1}     */
13         else
               /* The function F_i was already
               alone in the task τ_j       */
14             Restart the algorithm with another function
               chosen randomly;
15         end if
16     end if
       /* once the mutated solution is
       generated, we apply the assignment
       rules to generate the new
       composition of the task set         */
17     Apply the assignment rules on the new candidate
       solution to generate the new task set parameters
       (T_i,C_i,D_i) of each task;
       /* It remains to verify the
       schedulability of the new task set
       by simulation using the cheddar
       tool                                */
18     if The new task set is NOT schedulable then
19         Restart the algorithm with another function
           chosen randomly;
20     end if
21 end
```

a) Computation of the Task Period: We want to set a period for τ_i that releases the task as F_i and F_j should be released. First, we assume that the function F_j can be assigned to τ_i if and only if

$$\zeta_j\ mod\ T_i = 0 \quad or \quad T_i\ mod\ \zeta_j = 0.$$

Listing 1: Classical implementation of a periodic task with Ada

```
with Ada.Real_Time; use Ada.Real_Time;
...
task body Tau_i is
 -- Next_Time used for periodic suspension
 Next_Time : Time := Clock;
 Period    : constant Time_Span := Milliseconds(Ti);
begin
 loop
   -- calling the function run by the task  τi
   Fi;
   Next_Time := Next_Time + Period;
   -- Time-based activation event
   delay until Next_Time;
 end loop;
end Tau_i;
```

Listing 2: Ada implementation of the task τ_i containing two functions

```
with Ada.Real_Time; use Ada.Real_Time;
...
task body Tau_i is
 Next_Time          :Time := Clock;
 Number_Functions   : Integer := 2;
 Period_Functions   : Integer_Array (1..Number_Functions) := (ζi, ζj);
 Index_Functions    : Integer_Array (1..Number_Functions) := (i,j);
 Period             :constant Time_Span :=
                        Milliseconds(GCD(Period_Functions));
 Counter            : Integer := 0;
 Frequency          : Integer;
begin
 loop
  for i in 1 .. Number_Functions loop
    Frequency := Period_Functions(i)/Period;
    if (counter mod Frequency = 0) then
        Call_Function_by_index(Index_Functions(i));
    end if;
  end loop;
  counter :=
         (counter + 1) mod (LCM(Period_Functions)/Period);
  Next_Time := Next_Time + Period;
  delay until Next_Time;
 end loop;
end Tau_i;
```

This condition means that the function F_j is *harmonic* with task τ_i. Thanks to this assumption, the period of τ_i can be assigned as follows:

$$T_i = GCD(\zeta_i, \zeta_j) \tag{1}$$

With GCD is the Greatest Common Divisor of the periods of the functions F_i and F_j.

As a task may embed several functions, we need to provide a specific implementation of the task τ_i. This is described by the Listing 2. This implementation preserves the behavior of the mapped functions. Indeed, the added lines (displayed in **boldface**) to the implementation of the task ensure an internal scheduling of the contained functions according to their frequency.

Let consider the following example to illustrate the behaviour at run-time of several functions assigned in one task (Listing 2). We assume F_1 = (2, 5, 5) and F_2 = (1, 10, 10) assigned to a task τ.

According to the equation 1, the period of τ is equal to 5 ($GCD(10, 5) = 5$).

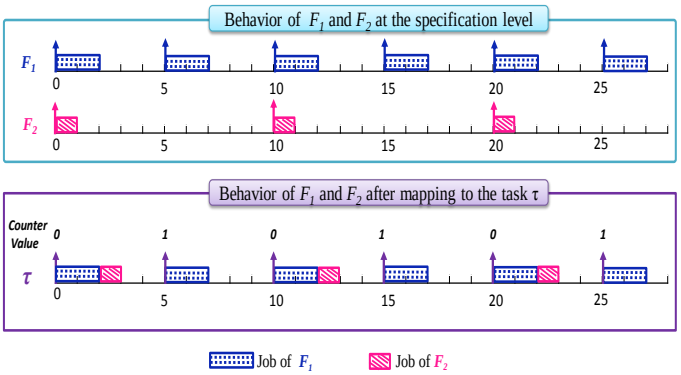

Fig. 5: Behavior of functions F1 and F2 before and after mapping

Figure 5 shows, first, a simulation of the execution of F_1 and F_2 assuming that each function is assigned to a task. Second, it shows also a simulation of their execution after mapping of both functions to the task τ. This figure shows how the implementation of Listing 2 as well as equation 1 enforce periodic activations of functions assigned to the task.

b) Computation of the Task Capacity: As shown in Figure 5, the execution time of a task τ_i containing two functions F_i and F_j varies from one period to another. However, we only consider the worst execution time of both functions run by the task. Therefore, the capacity of τ_i is set to the sum of the capacities of all functions contained in the task:

$$C_i = \gamma_i + \gamma_j \tag{2}$$

Obviously, it may reduce the schedulability of the mapping solution.

c) Computation of the Task Deadline: As a first approach, the deadline D_i of a task τ_i containing functions F_i and F_j is set as follows:

$$D_k = min(\delta_i, \delta_j) = \delta_i \quad with \quad \delta_i \leq \delta_j \tag{3}$$

This proposition may restrict the task τ_i with a smaller deadline than required by functions F_i and F_j. Again, it may also reduce the schedulability of the assignment solution.

By considering the mapping method described above, the constraint (i) is preserved. As we expect to also ensure the constraint (ii), in the sequel, we discuss how the mapping may jeopardize schedulability of the mapping solution.

D. Impact of the Mapping Method on the Schedulability

In the previous section (Section III-C), we have proposed rules for assigning functions to tasks. We have noted that the assumptions made in the mapping method may produce unschedulable task sets. In order to illustrate it, we consider a simple example of a functional model $\Gamma = \{F_1(1, 5, 5), F_2(3, 10, 10), F_3(3, 20, 20)\}$. The initial assignment model is given in Table I. It is schedulable (i.e., the corresponding processor utilization is equal to 65%) and then it is a feasible solution.

Another possible mapping is to assign F_3 in the task τ_1 as F_3 is harmonic with the task τ_1. The resulting task set will be as depicted in Table II. This mapping solution is not feasible.

TABLE I: Initial assignment model

$Task$	C	T	D
$\tau_1 = \{F_1\}$	1	5	5
$\tau_2 = \{F_2\}$	3	10	10
$\tau_3 = \{F_3\}$	3	20	20
CPU usage	65%		

TABLE II: A possible assignment model

$Task$	C	T	D
$\tau_1 = \{F_1, F_3\}$	4	5	5
$\tau_2 = \{F_2\}$	3	10	10
CPU usage	110%		

Since the mapping may affect the schedulability of the design model, we must check the schedulability of the mapping solutions explored by mutation.

Nevertheless, section IV of this article will show that the pessimistic deadline and capacity assignments may be negligible.

IV. Evaluation

In this section, we present experiments that show the effectiveness of our approach to generate, from the functional level, a set of design models. Each design model represents a trade-off between *low preemption costs* and *high laxity of tasks*.

In order to perform the experiments, we apply our PAES-based mapping method on synthetically generated function sets. Function periods ζ_i, are uniformly generated between 10 and 150 units of time. We have implemented the periods generator so that generated values are multiples of 10. Processor utilization factor U_i for each function F_i are tuned with the UUnifast algorithm [18], so that the function utilizations added up to the desired total utilization factor for the function set. Furthermore, the function deadlines are set to be equal to the periods ($\delta_i = \zeta_i$). The capacities γ_i are set based on the generated periods and processor utilization factors.

Experiments are conducted on a personal computer with a 2.4GHz Intel Core i7 processor, 8 GByte of memory and running Ubuntu 12.04 OS. We have implemented the PAES formulation of our approach in the Cheddar framework. This implementation is available in the Cheddar Repository[1]. In the following, we present results of both evaluations.

We perform two experiments. The first experiment aims at comparing the front found by the PAES-based method with the exact one for small-size test cases. Furthermore, in order to assess the effectiveness of the PAES-based method for larger sizes test cases, we perform a second evaluation that consists in investigating the quality of produced solution sets.

A. Experiment 1: Comparison with The Exact Method for Small-Size Test Cases

The first evaluation aims at comparing our PAES-based method with an exact method that determines the exact Pareto front through an exhaustive search among all function partitions. We have implemented the exact method as follow:

1. We enumerate all possible solutions of partitioning for a given function set size
2. We determine consistent solutions towards the defined assignment rules among all partitioning solutions
3. We determine valid solutions towards schedulability analysis among all consistent solutions
4. We compute the Pareto front solutions, according to the dominance concept, among valid solutions.

As it was already mentioned in the problem statement (Section I), the mapping problem is a combinatorial problem. An exhaustive search can be applied only for small-size test cases because the size of the search space is exponential with respect to the number of functions, i.e. the size of the test case.

We can handle, in a reasonable amount of time, test cases up to 11 functions. Table III shows a small-size test case of 11 functions, produced by our generator with a total processor utilization equal to 50%.

We first compute its exact Pareto front using the exact enumerative method described above, and we then also apply to it our PAES-based method, with 3000 PAES-iterations. The number of valid solutions examined by the PAES-based method is always lower than the number of performed iterations. Indeed, at each PAES-iteration the mutation procedure generates a valid solution that can be already investigated in previous PAES-iterations.

Task	Capacity	Period	Deadline
F_1	2	60	60
F_2	10	110	110
F_3	8	120	120
F_4	1	30	30
F_5	15	120	120
F_6	2	110	110
F_7	2	60	60
F_8	3	120	120
F_9	4	60	60
F_{10}	1	100	100
F_{11}	2	90	90

TABLE III: A small-size test case of 11 functions

For a test case of 11 functions, the number of all possible partitioning solutions is 678570 which is equal to the 11^{th} *Bell number* (B_{11}). For the considered 11-functions test case (Table III), among all possible partitioning solutions, the exact method finds 3508 consistent solutions towards the assignment rules. Among these consistent solutions, there are 2530 valid solutions (schedulable). 7 of those solutions are non-dominated, and constitute the Pareto front.

Figure 6 shows all the valid solutions found by the exact method as well as the Pareto front. For this test case, our PAES-based method finds the complete Pareto front. Furthermore, we have noted that the method succeeds in finding the exact front at the 226^{th} PAES-iteration. In the performed experiments, we run our method for a fixed number of PAES-iterations. However, this experiment shows that it should be interesting to investigate some convergence-based stopping criteria (which is out of the scope of this article).

[1] http://beru.univ-brest.fr/svn/CHEDDAR/trunk/src/framework/paes

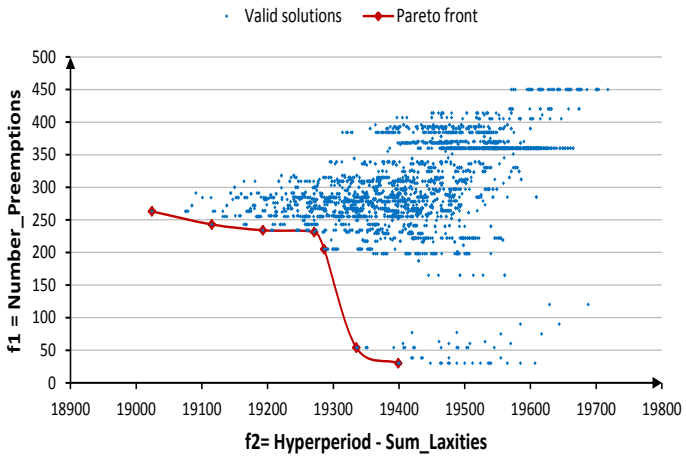

Fig. 6: Search space (valid solutions) and the exact Pareto front for the test case of 11 functions

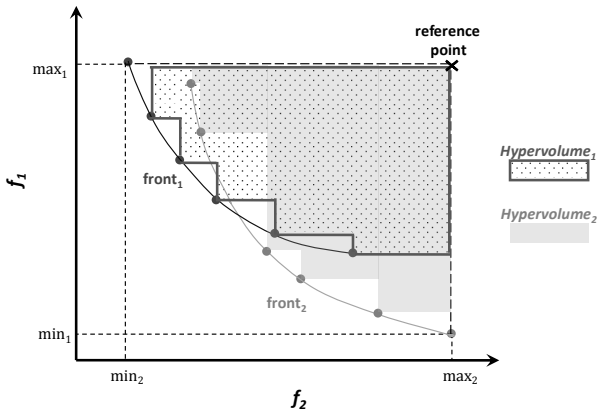

Fig. 7: Hypervolume performance indicator

This experiment shows the effectiveness of our method to determine the exact front (best trade-offs), for a small-size test case, in a very short time as compared to the exact method. Besides, although pessimistic choices we have made in Section III-C to determine parameters of tasks, experiments show that there are much schedulable solutions in the search space.

B. Experiment 2: Quality Evaluation using the Hypervolume Performance Indicator for Different System sizes

The set of solutions provided by our PAES-based method are to be compared qualitatively. The comparison between solution sets (fronts) is generally difficult since one set is not decidedly superior to another. Many *unary* metrics exist [19], that map a front to a single value thereby allowing to easily compare the quality of produced fronts. These metrics take into account both the quality of solutions obtained (accuracy) and their localization along the Pareto front range (diversity). The $hypervolume$ metric [19] is often used to compare at least 2 sets of solutions $front_1$ and $front_2$. It computes the area defined by each set, according to a reference point dominated by all of the points in the sets (figure 7). The set with the largest hypervolume is likely to present the best set of trade-offs.

For a test case, in order to compare different results among all runs, we compute the best and worst results obtained (considering non dominated solutions) for each objective (min_1, min_2), (max_1, max_2) over the set of solutions provided by all of the runs $\bigcup_{a \in runs} front_a$. Moreover, in order to allow the objectives to contribute approximately equally, values are normalized according to a linear normalization technique defined by [20]:
$(x, y) \in front_a \implies (x', y') \in norm_a$
with $x' = \frac{x - min_1}{max_1 - min_1}$ and $y' = \frac{y - min_2}{max_2 - min_2}$.
The reference point used is $(1 + \epsilon, 1 + \epsilon)$, in order to ensure that every point in $norm$ is dominated, with a small value of ϵ, e.g 0.001. The resulting hypervolume computed for one front can grow up to $1+\epsilon^2$ (in case when the front contains a single value, dominating all of the other values in fronts considered for comparison).

After normalizing fronts, the hypervolume indicator is computed using a dedicated tool provided by authors of [21].

A number of experiments were run in order to investigate the quality of solution sets for different function set sizes that range from 15 to 40 by step of 5 functions. In these experiments, the total processor utilization factor of generated function sets is fixed at 80%. For each size, we generate 10 different test cases (function sets) and each test case is processed with our PAES-based method 5 times because in the PAES algorithm there is a part of the random. The number of PAES-iterations for each run is fixed at 3000. For each test case, we compute the average of the hypervolume values of fronts produced by each of the corresponding runs. The hypervolume average values of test cases for each size are presented in Figure 8. The *cross* denotes the average hypervolume value over all test cases of the corresponding size and the ends of the vertical lines represent the maximum and minimum values.

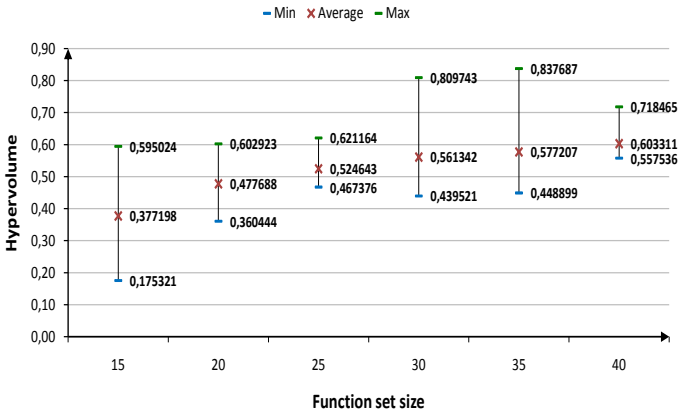

Fig. 8: Quality of solution sets for different function set sizes

This figure shows that the fronts corresponding to test cases with larger sizes ($\geq 25 functions$), having average values of hypervolume greater than 0.5, are more diversified than fronts of smaller size test cases. This result is quite logical. Indeed, all test cases of studied sizes are generated with the same total processor utilization and the same period interval.

Therefore, function capacities of larger test cases (size ≥ 25) are rather small compared to function capacities of smaller size test cases. This means that test cases of larger size have larger spaces of valid (i.e schedulable) solutions. Thus, fronts obtained for larger size test cases (size ≥ 25) are likely to be more diversified (and thus with shapes similar to the front in Figure 7), which could explain their high hypervolume values. The small hypervolume values (for test cases with size < 25) did not mean that the corresponding fronts are bad, but less diverse. For example, the hypervolume measured for the front shown in Figure 6 is 0.23, which is low value despite that the corresponding front is the exact front.

This hypothesis is reinforced by the average number of solutions in produced fronts : we can observe in the second row of Table IV that the number of solutions per front grows with the size of function sets.

To conclude, this shows that the PAES-based method makes effectively the architectural exploration and provides a set of promising trade-offs for the designer.

Function set size	15	20	25	30	35	40
Avg Nbr solutions	**5**	**9**	**12**	**15**	**13**	**14**
Avg std-dev	**0.00334**	**0.013308**	**0.036582**	**0.035009**	**0.043168**	**0.072582**

TABLE IV: Average number of solutions in produced fronts and average standard deviation of the hypervolume between runs

Furthermore, we have studied the stability of the PAES-based method between runs. For each size, we compute the standard deviation of the hypervolume between runs of each one of the considered test cases. The third row of Table IV represents the average standard deviation for each size. We can observe that the standard deviation increases when the size of test cases increases. This can be explained by the fact that the space of valid solutions is too large for the test cases with larger size and then the PAES-based method behave differently from one run to another for the same test case.

V. Related Work

In this section, we present the related works aiming at exploring and driving the design of real-time embedded systems. Then, we introduce some software architecture design approaches based on the clustering techniques.

In the literature, many approaches have been proposed to drive the design of architectural models that meet timing requirements. Indeed, software architecture optimization methods have been widely used to find solutions to the mapping problem. Authors of [22] provide a survey that classifies 188 publications according to various design objectives, constraints, addressed problem, degrees of freedom, used techniques, etc. However, none of the surveyed works considers both the number of preemptions minimization and the task laxities maximization as design objectives.

In [2] authors developed heuristic algorithms that generate, from a dataflow functional model with timing properties, the corresponding architectural model using Earliest Deadline First (EDF) for the scheduling analysis. The main objective of this work is to automatically make the functional-to-architectural mapping. However, the proposed method investigates only one architectural model and therefore it does not allow to explore and evaluate various architecture alternatives in order to optimize objective functions. Authors of [23] provide a framework for the integration and the development of real-time embedded systems. This framework allows designers to automatically generate, from a functional specification with dependency constraints described by the *Prelude* language a set of real-time tasks that can be executed on a uniprocessor architecture. Authors proved that the generated implementation enforces the system behavior as well as timing constraints described in the functional specification.

Again, this approach does not investigate architecture exploration, i.e. evaluation of many architecture solutions in order to optimize multi objective functions. Furthermore, a MARTE-based methodology was proposed in [24], enabling scheduling analysis at early stages of the software life cycle. It allows designers to generate, from a functional specification expressed with UML MARTE, a design model compliant with timing requirements. This work also does not address architectural exploration. Finally, in the distributed systems context, authors of [25] present a two step approach aiming at optimizing the deployment of real-time functions onto tasks with respect to end-to-end response time and latency constraints. In this work, authors perform architecture model exploration. However, in contrary to the work presented in this article, they do not focus on two contrary objective functions, i.e. related to both reducing the number of preemption and maximizing the task laxities. The problem is abstracted and resolved by different theories and optimization techniques, i.e. with mixed integer linear programming (MILP) and with genetic algorithm (GA).

In this article, we apply clustering techniques to propose and to evaluate several architecture solutions. However, clustering techniques are used in several works, in different contexts and for different purposes.

In [15], authors aim at minimizing overheads (timing and memory) of architecture composed of a large number of tasks by reducing the number of tasks through clustering. They propose an heuristic to automatically reduce the number of task while preserving real-time constraints. Contrary to the clustering approach proposed in our approach, the authors restrict clustering among tasks with identical periods.

Clustering tasks is also investigated in [26], in order to tackle the scheduling a large set of tasks with precedence constraints in a uniprocessor system. The authors propose to group tasks together according to their functional properties.

Furthermore, in order to ensure the refinement of the architectural model of an application to a specific real-time operating systems (RTOS), authors of [27] aim at reducing the number of priority levels used by a real-time application in order to comply with the number of priority levels provided by the targeted RTOS. They proposed a task clustering method based on MILP.

Finally, in the context of distributed systems, several approaches [28], [29], [30], [31] seek to reduce communications costs through clustering technique. In this context, clustering is applied as part of partitioning heuristics. They usually expect to minimize the overall execution time. A cluster can there be a set of tasks allocated to the same processor. Tasks are assigned

to clusters in order to ensure a schedulable task sets with minimum communication costs. In our approach, we do not focus on distributed as we assume uniprocessor architectures.

VI. Conclusion and Future Works

We presented in this article a design exploration method based on multi-objective optimization. We take into account two optimization objectives which are the minimization of preemption number and the laxity maximization. The addressed problem is quite difficult and our approach uses an approximating method as no exact method can be applied. Particularly, we define rules that drive the assignment of functions into tasks and we formulate the PAES MOO technique to explore possible assignment solutions. In order to, we define PAES specific operators in terms of the fitness functions, the encoding of solutions, and the mutation operator. We evaluated our approach with two ways. The first experiment shows the effectiveness of our PAES-based method in finding the best trade offs (i.e exact Pareto front) for small size case studies. For larger function sets, the second experiment shows that our method provides a set of promising trade-offs. As future works, we aim at extending our approach to address the mapping problem applied to functions with precedence constraints. Likewise, we expect to evaluate scalability of the proposed method with an industrial example composed of a high number of functions.

References

[1] F. Boniol, P.-E. Hladik, C. Pagetti, F. Aspro, and V. Jégu, "A framework for distributing real-time functions," in *Proceedings of the 6^{th} international conference on Formal Modeling and Analysis of Timed Systems*, F. Cassez and C. Jard, Eds., vol. 5215. Springer, 2008, pp. 155–169.

[2] C. Bartolini, G. Lipari, and M. Di Natale, "From functional blocks to the synthesis of the architectural model in embedded real-time applications," in *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, March 2005, pp. 458–467.

[3] G.-C. Rota, "The number of partitions of a set," *The American Mathematical Monthly*, vol. 71, no. 5, pp. 498–504, May 1964.

[4] J. D. Knowles and D. W. Corne, "Approximating the nondominated front using the pareto archived evolution strategy," *Evolutionary computation*, vol. 8, no. 2, pp. 149–172, 2000.

[5] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *SIGAda Ada Letters*, vol. 24, no. 4. ACM, 2004, pp. 1–8.

[6] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A practitioners handbook for real-time analysis: guide to rate monotonic analysis for real-time systems*. Springer US, 1993.

[7] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[8] J. Y.-T. Leung and M. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Information processing letters*, vol. 11, no. 3, pp. 115–118, 1980.

[9] S. Bandyopadhyay and S. Saha, "Some single- and multiobjective optimization techniques," in *Unsupervised Classification*. Springer Berlin Heidelberg, 2013, pp. 17–58.

[10] C. A. C. Coello, D. A. Van Veldhuizen, and G. B. Lamont, *Evolutionary algorithms for solving multi-objective problems*. Springer, 2002, vol. 242.

[11] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[12] E. R. Hruschka, R. J. G. B. Campello, A. A. Freitas, and A. P. L. F. De Carvalho, "A survey of evolutionary algorithms for clustering," *Transactions on Systems, Man, and Cybernetics Part C*, vol. 39, no. 2, pp. 133–155, 2009.

[13] T. Bäck, *Evolutionary algorithms in theory and practice : evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996.

[14] A. Thekkilakattil, A. S. Pillai, R. Dobrin, and S. Punnekkat, "Reducing the number of preemptions in real-time systems scheduling by cpu frequency scaling," November 2010, pp. 129–138.

[15] A. Bertout, J. Forget, and R. Olejnik, "Minimizing a real-time task set through task clustering," in *Proceedings of the 22^{nd} International Conference on Real-Time Networks and Systems*. ACM, 2014, pp. 23–31.

[16] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, Sep 1993.

[17] J. W. McCormick, F. Singhoff, and J. Hugues, *Building parallel, embedded, and real-time applications with Ada*. Cambridge University Press, 2011, vol. 1.

[18] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[19] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer Science & Business Media, 2007.

[20] C. M. Fonseca, J. D. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," in *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005)*, vol. 216, 2005, p. 240.

[21] C. M. Fonseca, L. Paquete, and M. López-Ibánez, "An improved dimension-sweep algorithm for the hypervolume indicator," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. IEEE, 2006, pp. 1157–1163.

[22] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *Software Engineering, IEEE Transactions on*, vol. 39, no. 5, pp. 658–683, 2013.

[23] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete event dynamic systems*, vol. 21, no. 3, pp. 307–338, 2011.

[24] C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard, "Optimum: a marte-based methodology for schedulability analysis at early design stages," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 1, pp. 1–8, 2011.

[25] A. Mehiaoui, E. Wozniak, S. Tucci-Piergiovanni, C. Mraidha, M. Di Natale, H. Zeng, J.-P. Babau, L. Lemarchand, and S. Gerard, "A two-step optimization technique for functions placement, partitioning, and priority assignment in distributed systems," *ACM SIGPLAN Notices*, vol. 48, no. 5, pp. 121–132, 2013.

[26] L. Santinelli, W. Puffitsch, A. Dumerat, F. Boniol, C. Pagetti, and J. Victor, "A grouping approach to task scheduling with functional and non-functional requirements," *Embedded Real-time Software and Systems (ERTS)*, Feb. 2014.

[27] R. Mzid, C. Mraidha, A. Mehiaoui, S. Tucci-Piergiovanni, J.-P. Babau, and M. Abid, "Dpmp: a software pattern for real-time tasks merge," in *Modelling Foundations and Applications*, 2013, vol. 7949, pp. 101–117.

[28] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, Apr 1995.

[29] L. Guodong, C. Daoxu, W. Daming, and Z. Defu, "Task clustering and scheduling to multiprocessors with duplication," in *Parallel and Distributed Processing Symposium*. IEEE, April 2003.

[30] A. Ahmadinia, C. Bobda, and J. Teich, "Temporal task clustering for online placement on reconfigurable hardware," in *IEEE International Conference on Field-Programmable Technology (FPT)*. IEEE, 2003, pp. 359–362.

[31] M. A. Palis, J.-C. Liou, and D. S. L. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *Transactions on Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46–55, January 1996.