Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 591983, 28 pages
doi:10.1155/2011/591983

Research Article

Static Scheduling of Periodic Hardware Tasks with Precedence and Deadline Constraints on Reconfigurable Hardware Devices

Ikbel Belaid,[1] Fabrice Muller,[1] and Maher Benjemaa[2]

[1] *LEAT-CNRS, University of Nice Sophia Antipolis, 06560 Valbonne, France*
[2] *Research Unit ReDCAD, National Engineering School of Sfax, University of Sfax, 3038 Sfax, Tunisia*

Correspondence should be addressed to Ikbel Belaid, ikbel.belaid@unice.fr

Received 27 August 2010; Revised 12 January 2011; Accepted 10 February 2011

Academic Editor: Michael Hübner

Task graph scheduling for reconfigurable hardware devices can be defined as finding a schedule for a set of periodic tasks with precedence, dependence, and deadline constraints as well as their optimal allocations on the available heterogeneous hardware resources. This paper proposes a new methodology comprising three main stages. Using these three main stages, dynamic partial reconfiguration and mixed integer programming, pipelined scheduling and efficient placement are achieved and enable parallel computing of the task graph on the reconfigurable devices by optimizing placement/scheduling quality. Experiments on an application of heterogeneous hardware tasks demonstrate an improvement of resource utilization of 12.45% of the available reconfigurable resources corresponding to a resource gain of 17.3% compared to a static design. The configuration overhead is reduced to 2% of the total running time. Due to pipelined scheduling, the task graph spanning is minimized by 4% compared to sequential execution of the graph.

1. Introduction

An important trend in real-time applications implemented in reconfigurable computing systems consists in using reconfigurable hardware devices to increase performances and to guarantee temporal constraints. These reconfigurable devices provide a high density of heterogeneous resources in order to satisfy application requirements and especially to enable parallel computing. Furthermore, the devices employ the pertinent concept of run-time partial reconfiguration which allows reconfiguration of a portion of available resources without interrupting the remainder parts running in the same device. Consequently, the concept increases resource utilization and application performance.

Periodic partially ordered activities represent the major computational demand in real-time systems such as real-time control and digital signal processing. This category of repetitive computation is described by directed acyclic graphs (DAGs). Implementation of these DAGs in reconfigurable hardware devices consists in scheduling tasks to a limited number of nonidentical units shaped on the area of reconfigurable resources, while respecting the four constraints described as follows. (1) The periodicity constraint: each task is repeated periodically according to its ready times in the graph. Thus, if task A has a period P_A, then for all $i \in \mathbb{N}$, $(r_{A_{i+1}} - r_{A_i}) = P_A$, where A_i and A_{i+1} are the ith and the $(i+1)$th repetitions of task A, and r_{A_i} and $r_{A_{i+1}}$ are their start times. (2) The precedence constraint: to maintain the rightness of task precedences, in each iteration, a task can be executed only if all its predecessors in the graph have finished their executions. Therefore, each task A must start execution after the completion of executions of its predecessors defined by the subset Π_A, thus for all $i \in \mathbb{N}$, $s_{A_i} \geq s_{B_i} + C_B$, for all $B \in \Pi_A$, where s_{A_i}, s_{B_i} are the start times of task A and task B, respectively, during their ith iteration, and C_B is the execution time of task B. (3) The dependence constraint: the execution of each task in DAG is launched when all the data resulting from all its predecessors are available. This constraint guides the choice of task periods as detailed in Section 3. (4) The deadline constraint: as this paper focuses on hard real-time systems, each task in the DAG must finish its execution before its hard deadline. Thus, within iteration i, if task A has an execution time C_A and an absolute deadline d_{A_i}, then $s_{A_i} + C_A \leq d_{A_i}$.

Figure 1 illustrates an example of the targeting task graph. As can be seen in Figure 1, the tasks are repeated according to their fixed periods. Each task with precedence link launches its execution only when its predecessors achieve their executions and only when it is required. For example, the third iterations of T_2 and T_3 of periods 8 do not need a third execution of their predecessor T_1 as it is less repetitive than T_2 and T_3 (period of T_1 is equal to 12). At each repetition, to enable the task execution, the dotted lines ensure the data transfer between interdependent tasks. The issue of data dependence is detailed later in the paper. Finally, at each iteration, the real-time tasks must respect their hard deadlines.

As shown in Figure 2, this paper proposes a new methodology comprising three main stages to achieve the scheduling of these DAGs with the predefined constraints on reconfigurable devices.

Task Clustering. This stage is technology dependent. It targets the partitioning of tasks requiring the same types of resources into the same cluster.

Mapping/Scheduling of Tasks in Clusters. This stage starts by performing spatial and temporal analyses mentioned in Figure 2 by DAG validity, Ready Times, and a set of heuristics. Subsequently, based on a predefined preemption model, it deals with simultaneous resolution of mapping tasks to the obtained clusters and global scheduling of tasks in clusters respecting the periodicity, precedence, dependence, and deadline constraints. This stage aims at optimizing scheduling quality.

Cluster Placement on the Reconfigurable Device. This stage is also technology dependent. It involves searching for the most suitable physical location partitioned on the reconfigurable device for each cluster obtained at the second stage. This stage aims at optimizing placement quality.

The resolution of these three stages results in static scheduling of tasks in the DAGs into a limited number of reconfigurable units partitioned on the device, respecting the periodicity, precedence, dependence, and deadline constraints. This is a fundamental problem in parallel computation, equivalent to determining static multiprocessor scheduling for DAGs in a software context. As is well known, static multiprocessor task graph scheduling is a combinatorial optimization problem, and it is formulated in this paper through mixed integer programming and solved by means of powerful solvers.

The paper details the spatial and temporal analyses required to check scheduling task graph feasibility and aims at determining the optimal solution in terms of schedule length, waiting time, parallel efficiency, resource efficiency, and configuration overhead. Schedulability analysis is not the focus of the present paper. However, before dealing with DAG scheduling on reconfigurable device, the rightness of the precedences and dependences between tasks within the graph and the accuracy of real-time functioning are analyzed, and a set of heuristics are performed to provide the number of reconfigurable physical units needed to ensure the existence of valid DAG scheduling. The analyses are expressed by some constraints to ensure the validity of the chosen task graph.

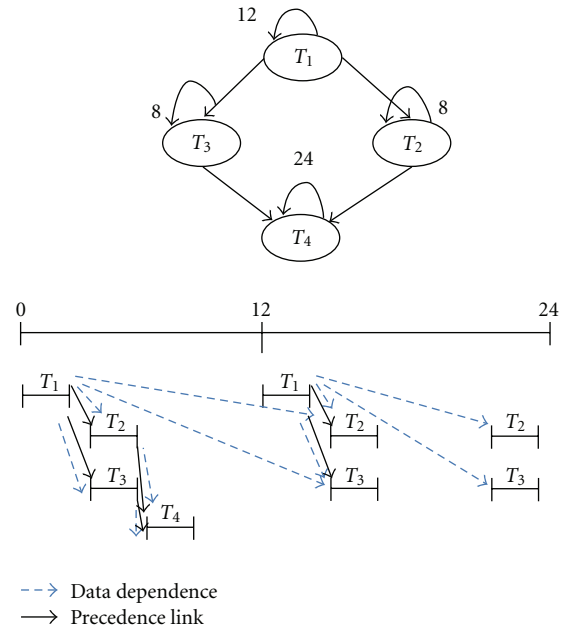


Figure 1: The targeted acyclic task graph.

The paper is organized as follows: Section 2 presents related works of the DAG scheduling problem. Section 3 details the methodology we propose to achieve the placement and scheduling of DAGs on reconfigurable devices. Section 4 describes an illustration of our proposed methodology on a given DAG and evaluates the obtained enhancements by metric measuring of placement and scheduling quality. The conclusion and future works are presented in Section 5.

2. Related Works

Static multiprocessor scheduling techniques using task graphs have matured over the last years, and many powerful scheduling strategies have emerged. As this problem is known to be NP-hard [1], the main research efforts in this area focus on heuristic methods and few of them propose analytic resolutions. We have studied static and dynamic multiprocessor scheduling using DAGs in both the software and hardware contexts.

In [2], Clemente et al. implement a static hardware scheduler employing efficient techniques which greatly reduce reconfiguration latencies and schedule length. Taking into account that configuration latency drastically reduces the efficiency of hardware multitasking systems, they introduce a new hardware scheduler communicating directly with the reconfigurable units and using optimization techniques: prefetch, reuse and replace while guaranteeing the precedence constraints. The prefetch technique manages in advance the reconfigurations and replacements required to improve task reuse. Reference [2] presents three algorithms

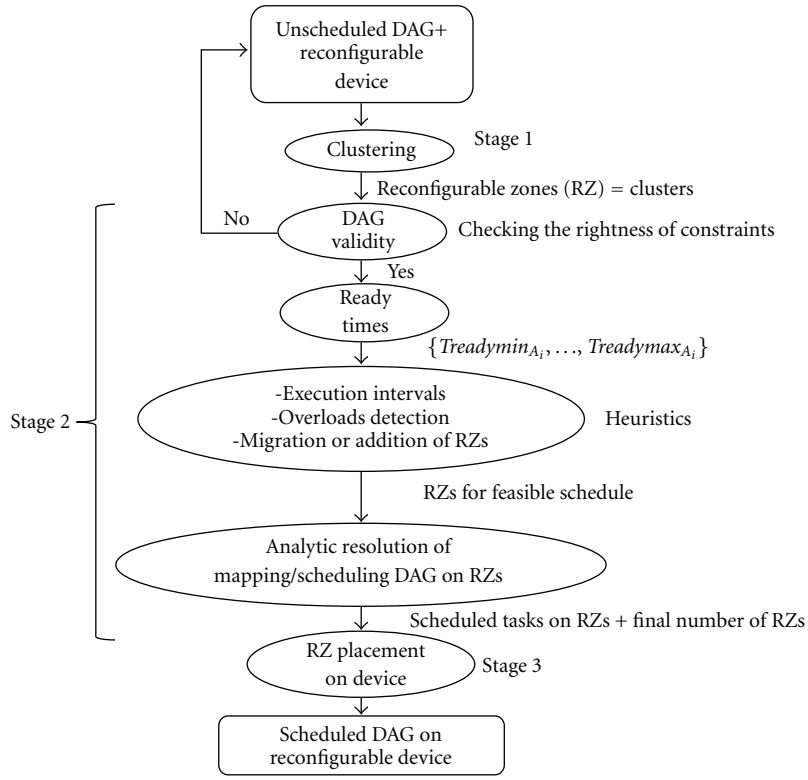


Figure 2: The proposed methodology.

for the replace technique, that is, the *Least Recently Used*, *Longest Forward Distance*, and *Look Forward + Critical* (*LF + C*) algorithms. The paper focuses especially on *LF + C* to schedule graph tasks on several reconfigurable units. In order to maximize the task reuse that reduces reconfiguration latency, *LF + C* classifies tasks from the most critical ones in terms of required reconfiguration to the least critical ones and tries to replace the latter tasks, so that their reconfiguration does not generate any overhead during task graph execution. This advantage is ensured by the prefetch technique which reconfigures a given task during execution of its predecessors.

Static multiprocessor scheduling is a very difficult problem, but genetic algorithms have successfully been applied to the search for acceptable solutions. In [3], the authors investigate scheduling for cyclic task graphs using genetic algorithms by transforming the cyclic graph into several alternate DAGs. To create an efficient schedule, the paper considers both the intracycle dependencies and the dependencies across different cycles. After unfolding the cyclic task graph for two cycles by incorporating the intercycle dependencies, the paper presents an algorithm investigating all the subgraphs extracted from a two-cycle task graph. Based on measurements such as the height and the width of the task graph, connection degree, degree of simultaneousness, and independent parts in the graph, the method evaluates the resulting subgraphs to select the configuration that best suits a chosen application and the available hardware configuration. Suitable allocation to processors is obtained by the *earliest start time* heuristic where tasks are assigned to the processor that offers the earliest start time. The employed genetic algorithm tries to optimize the schedule length which is expressed by the finishing time on all processors.

In [4], Abdeddaim et al. describe a method based on a timed automaton model for solving the problem of scheduling partially ordered tasks on parallel identical machines. The proposed method formulates for each task in the graph a 3-state automaton consisting of the waiting, active, and finish states. Therefore, by searching the tasks related by a partial order in the graph, the possible disjoint chains in the graph are extracted. The automaton of every chain is constructed using the individual task-specific automaton. The global automaton is then composed by the chain-specific automata and takes care that the transitions do not violate the precedence, and resource constraints. Thus, optimal scheduling consists in finding the shortest path in the timed automaton. The proposed methodology is also extended to include two additional features in the task-specific automaton which are the release and the deadline times.

Integer linear programming (ILP) formulation is exploited in some works of static multiprocessor scheduling using task graphs. The authors of paper [5] propose an exact ILP formulation to perform task graph scheduling on dynamically and partially reconfigurable architectures and that minimizes schedule length. In addition, this work proposes a dynamic reconfiguration-aware heuristic scheduler called *NAPOLEON*, which adopts an ALAP (as late as possible)

order of tasks and exploits configuration pre-fetching, module reuse, and antifragmentation techniques. Both methods are extended to the Hw/Sw codesign. The ILP formulation is based on nonpreemptive tasks, allows the execution of tasks on software processors or on the FPGA, and respects the FPGA physical constraints as well as the precedence and temporal constraints in the graph. Both methods provide a solution for the complete scheduling of the DAG and determine for each task its Sw or Hw execution unit, its time of reconfiguration start, its position on FPGA, and its execution starting time.

Another exact ILP formulation for performing task mapping and scheduling on multicore architectures is presented in [6]. The technique of these authors incorporates loop level task partitioning, task transformations by using loop fusion and loop splitting, and it aims at reducing system execution time. The paper focuses on an ILP-based approach for task partitioning, task mapping, and pipelined scheduling while taking data communication between processors into consideration for DSP applications on the multicore platform. The authors in [6] divide the problem into two parts. The first assigns and schedules tasks on processors by including the task merging on the same batch and the replication of batches to several processors. The second step conducts a mapping of data to memory architecture by minimizing memory access latencies.

In [7], Sandnes and Sinnen consider the scheduling of iterative computing that can be represented by cyclic task graphs. In order to avoid costly classic graph unfolding and to shorten the makespan during scheduling, the authors propose a new strategy for transforming cyclic task graphs into acyclic task graphs; an efficient scheduling from the literature named *Critical Path/Most Immediate Successor First* (*CP-MISF*), proposed by Kasahara and Narita in 1985, is then applied to the transformed graph. The strategy is based on a decyclification step involving three parts: (1) a decyclification algorithm for transforming the cyclic graph into an acyclic graph based on a given start node and depth first search (DFS) strategy, (2) by assuming that the critical path in the graph is a good estimator for its schedule length, it searches the start node that yields the shortest critical path in the transformed graphs, and (3) quantifying the acyclic graph quality in terms of makespan. In addition, based on an adjacency matrix representing the graph dependencies and simplifying the unfolding formulation, the paper presents a new intuitive graph unfolding formulation which decomposes the adjacency matrix into two matrices, one for intraiteration dependences and another for interiteration dependences. The unfolded graph is then scheduled using a genetic algorithm approach.

In [1], Djordjević and Tošić propose a new compile-time single-pass scheduling technique applied for task graphs onto fully connected multiprocessor architectures called *chaining* and which takes into account communication delays. The proposed technique consists in a generalized list scheduling with no preconditions concerning the order in which tasks are selected for scheduling. The main idea is to build an heuristic providing a trade-off between maximizing parallelism on processors, minimizing communication overheads, and minimizing overall execution time of the task graph. *Chaining* technique uses nonpreemptive tasks and constructs a scheduled task graph incrementally by scheduling one task at each step. The intermediate partially scheduled task graphs are obtained by selecting a nonscheduled task at each step and by placing it on the most appropriate precedence edge. The policy of selection of tasks to be scheduled is based on a *Task Selection First* heuristic, and the selection of the most suitable valid edge where the task will be placed is guided by the critical path and edge width criteria. The tasks encompassed within the same chain are scheduled on the same processor.

In [8], the authors aim at improving the performance of hardware tasks on the FPGA. Intertask communication and data dependences between tasks are analyzed in order to reduce configuration overhead, to minimize communication latency, and to shorten the overall execution of tasks. The work exploits the proposed works in reconfigurable computing and addressing resource efficiency to present three algorithms. *Reduced Data Movement Scheduling* (*RDMS*) is the most efficient dynamic algorithm for reducing configuration and communication overheads and provides the optimal performance for scheduling tasks in DAGs on the FPGA. *RDMS* uses the total reconfiguration of the FPGA and tries to minimize the number of reconfigurations by grouping communicating tasks in the same configuration. By conducting a width search, *RDMS* schedules tasks while respecting their data dependences. *RDMS* is based on dynamic programming algorithm and ensures that each configuration includes the combination of tasks that exploits the hardware resources to the maximum and that encompasses the highest possible number of task dependences.

In [9], Fekete et al. consider the optimal placement of hardware tasks in space and in time on the FPGA. Tasks are presented as three-dimensional boxes in space and time. The authors integrate intertask communication expressing data dependence and use a graph-theoretical characterization of the feasible packing determined by means of a decision of an orthogonal packing problem with precedence constraints. By searching the transitive orientations and by performing projections, the authors of paper [9] transform the 3D boxes representing tasks into 3×1D arrangements and then verify whether the three obtained arrangements referred to as packing classes satisfy the conditions of the feasible packing and determine the optimal spatial and temporal packing. This work enhances the makespan of the graph and optimizes the used reconfigurable space on the FPGA.

The major contribution in [10] is the development of a multitasking microarchitecture to perform a dynamic task scheduling algorithm on reconfigurable hardware for nondeterministic applications with intertask dependences which are not known until runtime. The task system is modeled as a modified directed acyclic graph which contains directed data edges and directed control edges labeled with scalar values indicating the probability of occurrence of the corresponding sink task in multiple task graph iterations [10]. Based on dynamic priority assignment for nonpreemptive tasks, the dynamic scheduler assigns each task to a software or hardware processing element, schedules

the contexts (bitstreams) and the data, and the fetch and the prefetch reconfigurations, and activates task execution. In order to minimize reconfiguration overhead, the dynamic scheduler uses the configuration prefetching technique to prefetch the task bitstream ahead of time or exploits the previous context configured in the logic cell. In addition, it aims to minimize application execution time by employing a local optimization technique.

In [11], Kohler defines a new heuristic to schedule DAGs on a system of independent identical processors. The author describes a simple critical path priority method which is shown to be optimal or near optimal in the most randomly generated computation graphs compared to the Branch and Bound method. This heuristic aims to minimize the finishing time of the computation graph. Critical path scheduling is based on a list (L) containing permutation of the tasks. Any time a processor is idle, it instantaneously scans the list L from the beginning and begins to execute the first free task which may validly be executed because all its predecessors have been completed [11]. The construction of the list is based on the critical path of tasks which is defined by the longest path from a given task to a terminal node. The paper also presents the exact Branch and Bound method used to obtain optimal scheduling, and the results obtained are compared to the critical path heuristic to prove the high quality of the latter method.

Table 1 provides a summary of the optimization parameters and employed techniques described in the cited works.

The major common drawback of most described techniques is that they do not address real-time constraints. Further-more, as shown in Table 1, most of them seek to optimize the makespan of the graph and neglect reconfiguration overhead and resource inefficiency or do not optimize the three parameters simultaneously. The works described in [8, 9] that conduct scheduling of DAGs on FPGA devices are based on successive total configurations of the device. Their resource efficiency consists only in the packing the maximum of tasks in the DAG on the FPGA in order to efficiently exploit the reconfigurable resources as well as to perform the minimum of total configurations. These works therefore do not consider the internal fragmentation caused by task placement on the FPGA which represents resource efficiency in our work.

In the context of hardware task scheduling, in the proposed works, the placement of scheduled tasks is not considered. Either the placement of the task is allowed in whatever position in the device (in this case, they do not take into account device heterogeneity) or the task is fixed to a unique reconfigurable unit which will reduce application flexibility. Contrary to these works, our strategy may be generalized for all types of devices, that is, both homogeneous and heterogeneous devices; the placement problem is considered an important stage, that is, highly interlinked with the scheduling of task graphs on the reconfigurable device. With our strategy, the task may be executed on several reconfigurable units according to its resources and according to the analyses conducted during the clustering stage.

Moreover, some of the described works do not exploit the relevant concept of run-time partial reconfiguration afforded by recent reconfigurable devices and employ the total configuration of FPGAs.

Based on these observations, our challenge is to utilize the benefits of the run-time partial reconfiguration concept for recent heterogeneous devices. The concept opens up the possibility of developing a hardware multitasking system by dividing the reconfigurable area into smaller reconfigurable units and by customizing them as required by the running application.

Considering multitasking, scheduling of task graphs on reconfigurable hardware devices is similar to heterogeneous multiprocessor scheduling in the software context. With full knowledge of the characteristics of DAG tasks and technology features, our methodology targets constrained applications and endeavors to provide pipelined scheduling in multi-reconfigurable-unit system while optimizing schedule length, waiting time, parallel efficiency, resource efficiency, and configuration overhead.

3. Proposed Methodology for Placement and Scheduling of Dags on Reconfigurable Devices

Our methodology can be viewed as two separate subproblems: (i) the mapping and scheduling of hardware tasks on predefined clusters by satisfying periodicity, precedence, dependence, and deadline constraints and (ii) the placement of obtained clusters on reconfigurable device taking into account its heterogeneity. Our resource and task management is essentially based on features of hardware tasks and reconfigurable hardware devices. The most recent Xilinx's Virtex FPGA was used as a reference for the reconfigurable hardware device to perform the placement and scheduling of the DAGs. Virtex SRAM-based FPGAs are characterized by a column-based architecture, a high density of heterogeneous resources, and several parallel configuration ports functioning at a high speed.

3.1. Terminology and Definitions. Throughout the paper, NT refers to the number of tasks in the graph, NZ the number of clusters, and NP the number of resource types in the chosen technology. The directed acyclic task graph is denoted by the pair (N, E). N is the set of nodes representing tasks in the DAG, and E is the set of edges linking the dependent tasks, $E \subseteq N \times N$.

As shown in Figure 3, on each edge, the outgoing value $(x_{A,B})$ from the source node A to the sink node B depicts the amount of data that A must produce at each repetition for B. The incoming value $(y_{B,A})$ represents the amount of data that must be consumed by the sink node B at each execution iteration after completion of the repetition of its predecessor A.

Each task in the graph has three models as follows.

3.1.1. Functional Model. Each hardware task (A) is represented by a set of parameters fixed at compile time and which are kept static throughout the DAG execution. A is characterized by its worst-case execution time (C_A), its

Table 1: Optimization parameters and techniques for scheduling works.

References	Makespan/Speedup/ Parallel efficiency	Resource efficiency	Configuration overhead	Techniques
[2]	x		x	Prefetch/replace/reuse of reconfigurations
[3]	x			Graph unfolding/genetic algorithm/*earliest start time* heuristic
[4]	x			Timed automaton model/shortest path
[5]	x		x	(1) ILP/reuse (2) *NAPOLEAN*/ALAP/prefetch/reuse/ antifragmentation
[6]	x			ILP/loop level task partitioning/task transformation (loop fusion + loop splitting)/task mapping and scheduling (task merging on batches + batch replication)/data mapping
[7]	x			(1) Graph decyclification (DFS + shortest critical path)/*CP-MISF* (2) Graph unfolding/genetic algorithm
[1]	x			*Chaining*/*task selection first* heuristic/critical path/edge width/communication latencies
[8]	x	x	x	Dynamic programming algorithm/FPGA total configuration
[9]	x	x		Orthogonal packing problem/packing classes
[10]	x		x	Prefetch/local optimization/reuse
[11]	x			Critical path heuristic/Branch and Bound

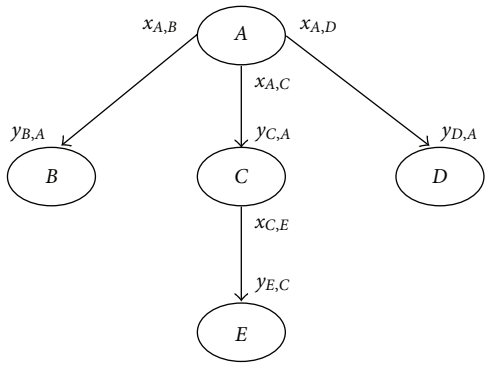


Figure 3: Directed acyclic graph.

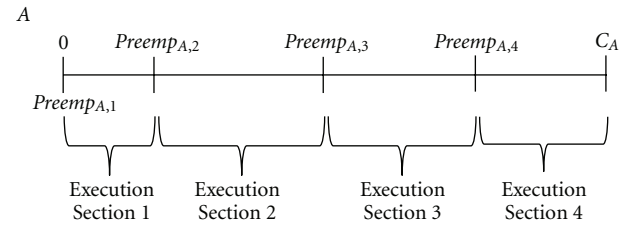


Figure 4: Predefined preemption points in Task A.

period (P_A), which is equal to its relative deadline (D_A), and a set of preemption points ($Preemp_{A,l}$). The preemption points are instants of the time taken throughout the worst-case execution time as shown in Figure 4. The number of preemption points of task A is denoted by $NbrPreemp_A$. This number also includes the first point of execution of the task. The set of preemption points is determined by the designer according to the known states in the behavioral model and according to possible data dependences between these states. The predefinition of preemption points gives rise to the execution sections within the hardware task. Our methodology is based on preemptive modeling to create a reactive system, to increase flexibility towards application needs, and consequently to enhance the respect of real-time constraints.

3.1.2. Behavioral Model. This includes the finite state machine controlling each task and which handles a set of registers or a small memory bank useful for context switching during preemption. The latency required to preempt and to resume the execution of tasks is disregarded as in the worst case; the time to access the system bus and memory is negligible. With the preemptive model, we do not use the classical method of readback and load modified bitstream since latency with this method is significant; it complicates preemption and requires a large space memory. With the classical method, a new readback bitstream must be saved at each preemption. With our model, the number of preemptions within tasks is limited by specifying the

possible preemption points according to task states outside of which preemption is not allowed. Thus, we resort to saving/loading the current state of the finite state machine with an acceptable amount of data by maintaining the same bitstream for each task within a given reconfigurable unit when the task needs to be preempted or resumed. Preemption points of hardware tasks are set in a way that reduces the data dependences that could exist between two states. In fact, maintaining a preemption point between two states processing the same data must be avoided because these data need to be saved into an external memory which might increase the preemption overhead at runtime. Otherwise, it is recommended to insert a preemption point when the task is in a blocked state, waiting to receive an external resource to enable the ready tasks to be executed in the reconfigurable unit. In the finite state machine, the longest execution time between two states must be considered in order to deduct the worst-case execution time.

3.1.3. RB-Model. At the physical level, tasks are presented as a set of reconfigurable resources called reconfigurable blocs (RB). RBs correspond to the physical resources in the reconfigurable hardware device required to achieve execution of the hardware task, and they define the RB model of the task as expressed by (1). Determination of the RB-model of hardware tasks is well detailed in our work described in [12]. The number of RB types is equal to the number of resource types in the chosen technology. The RBs are the smallest reconfigurable units in the hardware device. They are determined according to the available reconfigurable resources in the device, and they closely match its reconfiguration granularity. Each type of RB is characterized by a specified cost, $RBCost_k$, defined according to its frequency in the device, its power consumption, and the importance of its functionality,

$$A\_\mathrm{RB} = \{\alpha_{A,k}\mathrm{RB}_k\}, \quad \alpha_{A,k} \in \mathbb{N}, \quad 1 \le A \le \mathrm{NT}, \quad 1 \le k \le \mathrm{NP}. \tag{1}$$

The reconfigurable device is also characterized by its RB model as shown in our work described in [12] to enable the placement of hardware task clusters at a later stage.

The three main stages of the methodology used for static scheduling of DAGs on multi-reconfigurable-unit system with predefined constraints are described below.

3.2. Hardware Task Clustering. This stage comprises two steps that are performed consecutively: (i) reconfigurable zone type search and (ii) cost D computing. Bearing in mind that the concept of run-time partial reconfiguration had to be used, our main objective at this first stage was to partition tasks constituting the graph into cluster types determined according to their required RB types in order to enhance resource utilization.

3.2.1. Reconfigurable Zones Types Search. This step takes as input the RB model of each task in the DAG, and by performing Algorithm 1 of the worst-case complexity $O(\mathrm{NT}^2 * \mathrm{NP})$, it groups tasks sharing the same types of RBs under the same type of cluster by taking the maximum number of RBs between these tasks. With our methodology, the obtained types of clusters are denoted as reconfigurable zones (RZs). The upper bound of the possible RZs is NT. Thus, RZs are virtual units customized by Algorithm 1 to model the classes of hardware tasks in terms of RB types. RZs separate hardware tasks from their execution units on the reconfigurable device. In the last stage of our proposed methodology, RZs will be placed on their suitable reconfigurable units respecting the heterogeneity of the device and optimizing resource efficiency as well as configuration overhead. After the completion of Algorithm 1, each RZ is represented by its RB model as expressed by

$$\mathrm{RZ}_j\_\mathrm{RB} = \left\{\beta_{j,k}\mathrm{RB}_k\right\}, \quad \beta_{j,k} \in \mathbb{N}, \quad 1 \le j \le \mathrm{NZ}, \quad 1 \le k \le \mathrm{NP}. \tag{2}$$

Algorithm 1 processes the tasks of the DAG as follows. It scans the RB model of each hardware task and checks whether an already inserted type of RZ that closely matches the required types of RBs in the task exists in the RZ types list, *list-RZ* (line 6). Should this be the case, Algorithm 1 updates the number of RBs within this type of RZ by the maximum between the number of RBs in the task and that in the RZ (line 9).

If the required types of RBs in the task do not match any type of RZ included in the *list-RZ*, the algorithm of the search of RZ types decides on the creation of a new type of RZ as required by the task (lines 12, 13) and inserts it in *list-RZ* (line 14).

Figure 5 is an example of the execution of Algorithm 1 for the RZ types search for DAG comprising five tasks. Figure 5 illustrates the search for RZ types resulting from five tasks in a technology including four types of RBs (RB_1, RB_2, RB_3 and RB_4). A and C are grouped in the same type of RZ (RZ_1) as both need RB_1 and RB_2, and the number of each RB type within RZ_1 is adjusted by the maximum number of RBs between A and C. Similarly, RZ_2 is created by B and D, and E defines the third type of RZ (RZ_3).

After searching for the set of RZs, the configuration overhead for each obtained RZ is computed and denoted by $Config_j$. $Config_j$ corresponds to the configuration overhead to fit RZ_j in the target technology. This configuration overhead is computed by the floorplanning of each RZ_j on the chosen device and by conducting the whole partial reconfiguration flowup to the creation of the partial bitstream. $Config_j$ is determined by (3) according to configuration frequency and configuration port,

$$Config_j = \frac{\text{size of bit stream}}{(\text{Configuration frequency} \times \text{configuration port width})}. \tag{3}$$

3.2.2. Cost D Computing. This step commences by computing cost D between tasks and each RZ type resulting from the

(1) $RZ\text{-}type = 0$ // RZ types
(2) $List\text{-}RZ$ // list of RZ types
(3) n // natural
(4) **for all** tasks A **do**
(5) // $A\_\mathrm{RB} = \alpha_{A,k}\mathrm{RB}_k$
(6) **if** $((RZ\text{-}type \neq 0)$ **and** $(\exists n, 1 \leq n \leq RZ\text{-}type)/\forall k((\alpha_{A,k} \neq 0$ **and** $\beta_{n,k} \neq 0)$ **or** $(\alpha_{A,k} = 0$ **and** $\beta_{n,k} = 0)))$ **then**
(7) // this test checks whether the task matches with an RZ type that already exists in $list\text{-}RZ$
(8) **for all** k **do**
(9) $\beta_{n,k} = \max(\alpha_{A,k}, \beta_{n,k})$ // update RB number of RZ_n
(10) **end for**
(11) **else**
(12) Increment $RZ\text{-}type$
(13) $\mathrm{RZ}_{RZ\text{-}type} = Create\ new\ \mathrm{RZ}(\alpha_{A,k})$ // new type of RZ, $\mathrm{RZ}_{RZ\text{-}type} = \{\alpha_{A,k}\mathrm{RB}_k\}$
(14) Insert $(list\text{-}RZ, \mathrm{RZ}_{RZ\text{-}type})$
(15) **end if**
(16) **end for**

Algorithm 1: RZ types search or hardware task classes search.

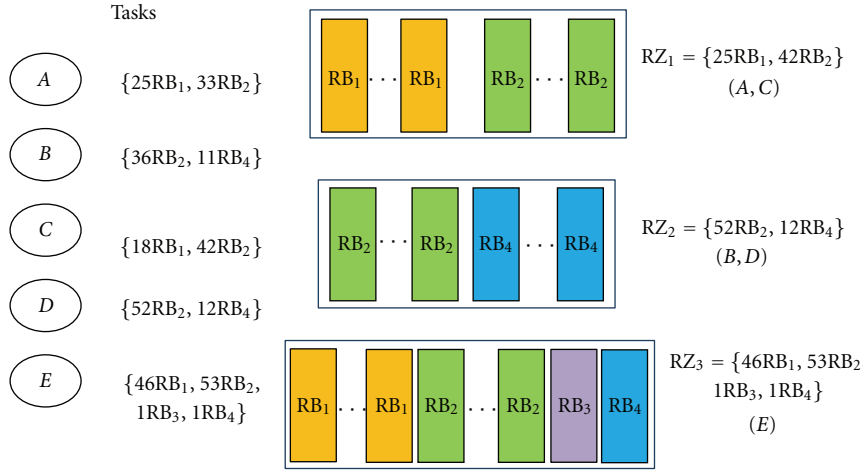


Figure 5: Example of the search for RZ types.

first step. Cost D represents the differences in RBs between tasks and RZs; consequently, it expresses resource wastage when a task is mapped to an RZ. Based on RB models of task A and RZ RZ_j, cost D is computed according to two cases as follows. Firstly, we define by

$$d_{A,j,k} = \alpha_{A,k} - \beta_{j,k}, \quad 1 \leq A \leq \mathrm{NT}, \quad 1 \leq j \leq \mathrm{NZ}, \quad 1 \leq k \leq \mathrm{NP}. \tag{4}$$

Case 1. For all k, $d_{A,j,k} \leq 0$, RZ_j contains a sufficient number of each type of RB (RB_k) required by A, and cost D is equal to the sum of the differences in the numbers of each RB type between A and RZ_j weighted by $RBCost_k$ as expressed in

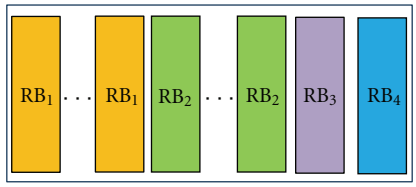
$$D\left(A, \mathrm{RZ}_j\right) = \sum_{1 \leq k \leq \mathrm{NP}} RBCost_k \times \left| d_{A,j,k} \right|. \tag{5}$$

Case 2. There exists k, $d_{A,j,k} > 0$, the number of RBs required by A exceeds the number of RBs in RZ_j or A needs RB_k which is not included in RZ_j. In this case, cost D is infinite,

$$D\left(A, \mathrm{RZ}_j\right) = \infty. \tag{6}$$

Cost D is exploited during the stage of task mapping and scheduling on RZs and the RZ placement stage to optimize the utilization of costly resources to the device. The execution of a given task in an RZ is allowed only when the cost D between them is finite. Figure 6 illustrates the computing of costs D between the five tasks and RZ_3 described in Figure 5.

3.3. Mapping and Scheduling of Hardware Tasks on RZs. It is well known that task mapping and scheduling are highly interdependent. The two issues therefore need to be addressed together for mapping and scheduling to be efficient. In order to analyze the scheduling of a given DAG



$RZ_3 = \{46RB_1, 53RB_2, 1RB_3, 1RB_4\}$ (E)

$RBCost_1 = 20$, $RBCost_2 = 80$, $RBCost_3 = 192$, $RBCost_4 = 340$

$D(A, RZ_3) = 20 \times |25\text{–}46| + 80 \times |33\text{–}53| + 192 \times |0\text{-}1| + 340 \times |0\text{-}1|$

$D(B, RZ_3) = \infty$ (Lack of $10RB_4$ in RZ_3 for B)

$D(C, RZ_3) = 20 \times |18\text{–}46| + 80 \times |42\text{–}53| + 192|0\text{-}1| + 340 \times |0\text{-}1|$

$D(D, RZ_3) = \infty$ (Lack of $11RB_4$ in RZ_3 for D)

$D(E, RZ_3) = 0$

Figure 6: Example of computing cost D with RZ_3.

of periodic tasks, it is sufficient to study its behavior for a time interval equal to the least common multiple of all the task periods, called the hyperperiod (HP). Consequently, the possible iterations of execution of each task A during the HP may be determined according to its period P_A, which is equal to $\lfloor HP/P_A \rfloor$. To resolve the subproblem of mapping and scheduling of hardware tasks on RZs, our methodology conducts three steps of spatial and temporal analyses. The first one checks the rightness of task precedences, dependences, and real-time functioning in the DAG by means of three essential constraints. Consequently, the DAG will be validated to launch the following analyses. The second analysis determines the lists of ready times of each task for each possible iteration during the hyperperiod. These lists take into account the periodicity, precedence, dependence, and deadline constraints. The third straightforward analysis takes as input the lists of ready times, searches at each iteration the possible execution intervals for each task, and therefore detects possible conflicts due to overlapping between execution intervals of parallel tasks on the same RZ. The third analysis pursues its processing to solve the detected overloads within RZs by performing either the migration of execution sections of some tasks respecting their predefined preemption points or by increasing the number of overloaded RZs.

3.3.1. Checking of Precedence, Dependence, and Real-Time Rightness in DAG. The first temporal analysis does not take into account spatial constraints and considers that there is an available RZ for each task. It also considers the periodicity of tasks. The main objective of this analysis is only to validate the correctness of task precedence and dependences and real-time constraints in the studied DAG. It is conducted by means of three essential constraints.

(a) Dependence Checking. More than precedence, we consider that the tasks related by an edge of precedence are also dependent. This means that the execution of a given task requires the data resulting from the execution of all its predecessors. This dependence is expressed by (7). Equation (7) focuses on periods and the amount of interchanged data between dependent tasks. It guarantees, when $P_A \geq P_B$ and $A \in \Pi_B$, that each data $(x_{A,B})$ produced by task A in its repetition A_i is consumed by its successor B during its iterations of execution B_i or $B_{j>i}$. When $P_A \leq P_B$ and $A \in \Pi_B$, at each iteration of execution of B, it ensures that there are the sufficient data $(y_{B,A})$ for B to be executed.

$$y_{B,A} \times \left\lfloor \frac{HP}{P_B} \right\rfloor = x_{A,B} \times \left\lfloor \frac{HP}{P_A} \right\rfloor, \quad \forall B, A \in \pi_B. \tag{7}$$

The previous equation eliminates the problem of data wastage and ensures that the data produced by all the iterations of each task are consumed by all the iterations of its successors. In this work, we focus on the case where $P_A \geq P_B$, for all $B, A \in \pi_B$.

(b) Precedence Checking. As we work in the case of $P_A \geq P_B$, for all $B, A \in \pi_B$, considering the periodicity constraint, this constraint claims that each iteration of execution A_i of a given task A may include only the iterations B_i or $\{B_{j<i}\}$ of its successors $\{B\}$ to ensure the correctness of the precedence constraint. During repetition A_i of a given task A, this constraint prohibits that B_i and $\{B_{j>i}\}$ repetitions of its successors $\{B\}$ coexist. In this way, during the HP, each ith execution of any task is preceded by the ith execution iterations of all its predecessors. To guarantee these rules, the following constraint expressed by (8) was developed to guide the selection of execution times and periods for tasks in the DAG:

$$\begin{aligned} &Tready_B + k \times P_B > Tready_A + (k-1) \times P_A, \\ &\forall B, A \in \pi_B, \quad k \in \mathbb{N}, \quad 1 \leq k \leq NbrIter_A. \end{aligned} \tag{8}$$

$NbrIter_A$ depicts the possible number of execution iterations of A during the HP. $Tready_A$ and $Tready_B$ are the ready times for tasks A and B without considering the spatial constraints. They are determined by searching the critical path corresponding to the task in the graph by using the ASAP (as soon as possible) technique. For each task B having a set of predecessors $\{A\}$, its ready time $Tready_B$ is computed as follows

$$\begin{aligned} Tready_B &= \max_{\{A \in \pi_B\}} \left(Tready_A + C_A \right), \\ Tready_B &= 0, \quad \text{if } \pi_B = \varnothing. \end{aligned} \tag{9}$$

(c) Real-Time Checking. Considering the periodicity constraint, this constraint analyses the respect of real-time constraints in the best case of spatial conditions in terms of number of RZs. By respecting the precedence and dependence constraints, (10) checks at each iteration whether each task may complete its execution before its strict absolute deadline. If the absolute deadline of the task for its last

iteration exceeds the HP, the deadline then turns into the HP. As we work in the case $P_A \geq P_B$, for all $A \in \pi_B$, the second expression of (10) satisfies the deadline constraint for the remaining repetitions of task B when its predecessors $\{A\}$ achieve all their execution iterations

$$\begin{aligned}
&\max\left(Tready_A + k \times P_A + C_A, Tready_B + k \times P_B\right) + C_B \\
&\quad \leq \min\left(Tready_B + (k+1) \times P_B, \mathrm{HP}\right), \quad \forall B, A \in \pi_B, \\
&\qquad k \in \mathbb{N}, \quad 0 \leq k \leq NbrIter_A - 1, \\
&Tready_B + k \times P_B + C_B \leq \min\left(Tready_B + (k+1) \times P_B, \mathrm{HP}\right) \\
&\forall B, k \in \mathbb{N}, \quad \max_{A \in \pi_B}(NbrIter_A) \leq k \leq NbrIter_B - 1.
\end{aligned} \tag{10}$$

Respect of the three previous constraints validates the selection of periods and execution times for periodic tasks in the graph for the scheduling with precedence, dependence constraints, and under strict real-time constraints on an unlimited number of reconfigurable units. Nevertheless, the following temporal and spatial analyses will extract the corresponding number of RZs that will satisfy these constraints. When the previous constraints are unreal, the DAG is considered invalid and consequently it will be rejected or the features of its tasks will be modified and evaluated again till it respects the constraints expressed by (7), (8), and (10).

Consequently, with our methodology, we depart from the temporal analysis to construct a suitable physical architecture allowing a feasible schedule. As a final step at this stage, after fixing the physical architecture of the multi-reconfigurable-unit system, analytic resolution of mapping/scheduling provides the optimal scheduling of DAGs on target technology.

3.3.2. Determination of Lists of Ready Times. The temporal analysis does not consider the physical constraints and searches all the ready times in each execution iteration for all tasks in the graph by respecting the precedence, dependence, periodicity, and real-time constraints. This analysis yields, by means of (11), the lists $\{Treadymin_{B_i}, \ldots, Treadymax_{B_i}\}$ during which task B may start execution, where B_i denotes the ith iteration of task B within the HP. Outside this list, task B might not respect the predefined constraints, which would lead to unfeasible scheduling.

For each task B, $Treadymin_{B_i}$ is the lower bound time to start task execution at its iteration i in order to respect its precedence, dependence, and periodicity constraints. $Treadymax_{B_i}$ is the upper bound time from which the task can no longer start execution if its strict deadline and the data dependency and precedence of its successors are to be respected. To compute the *Treadymin* of tasks, we start computing from the top of the DAG. For the *Treadymax* calculation, we start from the bottom of the DAG, and for both, we proceed using the breadth-first search strategy

$$\begin{aligned}
&Treadymin_{B_1} \\
&\quad = 0 \quad \text{if } \pi_B = \varnothing, \\
&Treadymin_{B_i} \\
&\quad = \max\left(\max_{A \in \pi_B}\left(Treadymin_{A_i} + C_A\right), rmin_{B_i}\right), \\
&Treadymax_{B_1} \\
&\quad = \min\left(\min_{A \in \varphi_B}\left(Treadymax_{A_1}\right), P_B\right) - C_B, \\
&\qquad \varphi_B \text{ is the set of successors of } B, \\
&Treadymax_{B_i/i>1} \\
&\quad = \min\left(\min_{A \in \varphi_B}\left(Treadymax_{A_i}\right), rmax_{B_{i+1}}, \mathrm{HP}\right) - C_B.
\end{aligned} \tag{11}$$

$rmin_{B_i}$ and $rmax_{B_i}$ are the start times of the ith repetition of B according to its first ready times ($Treadymin_{B_1}$ and $Treadymax_{B_1}$). Hence, they are determined by incrementing $Treadymin_{B_1}$ and $Treadymax_{B_1}$ by P_B. For example, if we have a task B with a period = 8, a hyperperiod HP = 24, and we consider $Treadymin_{B_1} = 3$ and $Treadymax_{B_1} = 4$, then $rmin_{B_1} = Treadymin_{B_1} = 3$, $rmin_{B_2} = 3 + 8 = 11$, $rmin_{B_3} = 11 + 8 = 19$, $rmax_{B_1} = Treadymax_{B_1} = 4$, $rmax_{B_2} = 4 + 8 = 12$, $rmax_{B_3} = 12 + 8 = 20$.

3.3.3. Determination of Task Execution Intervals and the Number of RZs. This temporal and spatial analysis considers the RZ types resulting from the task clustering stage and searches the possible parallelism between tasks to study execution conflicts on the RZs. When an overload is detected in some RZs, the analysis starts by solving this problem through a migration mechanism; if migration does not produce a solution, it increments the number of overloaded RZs as required. This analysis searches first, by means of Algorithm 2, the execution intervals of each task for each possible iteration during the HP, then, using Algorithm 3, it deals with the overlapping execution intervals on the RZs to search the possible overloads, and finally, it uses Algorithm 4 to try to solve the found overloads by the migration mechanism; when this latter mechanism fails, additional RZs are inserted in the architecture of the multi-reconfigurable-unit system to solve the persisting overloads.

(a) Search of Execution Intervals. This step uses the functional model of tasks and determines their execution intervals by means of Algorithm 2 of worst-case temporal complexity equal to $O(\mathrm{NT} * \mathrm{HP}^3)$. An execution interval for a given task at a given iteration is an interval during which the task can be executed while satisfying all the predefined constraints.

For each task A in the DAG, Algorithm 2 produces the set of its possible execution intervals expressed by

(1) A, B // Tasks
(2) i // Natural, iterations of execution of task A
(3) $List\text{-}Tready_{A_i}$ // the list of ready times for task A during ith iteration
(4) $Tready_{A_i}$ // the ready times for task A during ith iteration
(5) $Tready_A$ // the ready times for task A from the current $Tready_{A_i}$
(6) $Execution\text{-}Interval_{A_i} = \oslash$ // the set of execution intervals for task A during ith iteration
(7) **for all** tasks A **do**
(8) **for all** execution iterations i of task A **do**
(9) $List\text{-}Tready_{A_i} = \{Treadymin_{A_i}, \ldots, Treadymax_{A_i}\}$
(10) **if** $i = 1$ **then**
(11) $Execution\text{-}Interval_{A_i} = \{[Tready_{A_i}, \min(Tready_{A_i} + P_A, \min_{B \in \varphi_A}(Treadymax_{B_i}), \mathrm{HP})], \forall Tready_{A_i} \in List\text{-}Tready_{A_i}\}$
(12) **else**
(13) **for all** $Tready_{A_1} \in List\text{-}Tready_{A_1}$ **do**
(14) $Tready_{A_i} = \mathrm{First}\ (List\text{-}Tready_{A_i})$
(15) **for all** $Tready_A \in \{Tready_{A_i}, \ldots, Treadymax_{A_i}\}$ **do**
(16) $Execution\text{-}Interval_{A_i} = Execution\text{-}Interval_{A_i} \bigcup \{[Tready_A, \min(Tready_{A_1} + i * P_A, \min_{B \in \varphi_A}(Treadymax_{B_i}), \mathrm{HP})]$, if $Tready_{A_1} + i * P_A$ is chosen as upper bound for this interval then it must respect: $Tready_A + C_A \leq Tready_{A_1} + i * P_A$, if HP is chosen then it must respect: $Tready_A + C_A \leq \mathrm{HP}\}$
(17) **end for**
(18) $List\text{-}Tready_{A_i} = \mathrm{next}\ (List\text{-}Tready_{A_i})$
(19) **end for**
(20) **end if**
(21) **end for**
(22) **end for**

Algorithm 2: Search of execution intervals.

$Execution\text{-}Interval_{A_i}$ at each possible iteration i during the HP. When this algorithm processes the first iteration of task A, the set of its possible execution intervals is determined directly (line 11) considering the precalculated ready times in $List\text{-}Tready_{A_1}$, and the periodicity and dependence constraints. For the next iterations, considering each possible $Tready_{A_1}$ for the purpose of respecting the periodicity constraint (line 13), at each iteration i, Algorithm 2 searches all the corresponding execution intervals starting with each possible $Tready_{A_i}$ (line 14, line 15), and considering the dependence and the deadline constraints, and the HP to determine the upper bound for each execution interval (line 16). In line 18, in order to guarantee the periodicity and dependence constraints, progressing from a $Tready_{A_1}$ to another in list $List\text{-}Tready_{A_1}$, Algorithm 2 must shift the list $List\text{-}Tready_{A_i}$ by omitting its first element, since ready times in each iteration i are also linked to the ready times of the first iteration.

(b) Search of Overlapping between Execution Intervals of Tasks in the Same RZ. For the first iteration, the parallel tasks on a given RZ are extracted directly from the DAG; hence, there is no parallel execution between a given task and its successors. However, in the next iterations i, searching parallel tasks must respect some rules. For the next iterations i, for parallel efficiency, a given task could overlap with its successors during their iterations j ($j < i$) on the same RZ. Moreover, this step must also consider the execution conflicts on the same RZ for a given task in its iteration i and other tasks that are in the iterations j ($j \leq i$) when there are no dependence constraints between them. This step prohibits simultaneous execution of several iterations of the same task.

The step defines for each task the possible RZs allowing its execution in terms of types and number of RBs based on computed cost D and then searches the execution conflicts on the RZs using Algorithm 3. Algorithm 3 has a worst-case temporal complexity equal to $O(\mathrm{NZ} * 2^{\mathrm{NT} * \mathrm{HP}})$.

During each iteration i, for each RZ, in order to find the parallel tasks with a finite cost D with the RZ, Algorithm 3 searches all the possible combinations of sets of execution intervals $\{Execution\text{-}Interval_{A_j}\}$ of all the tasks that can be executed on the current RZ and produce overlapping execution intervals respecting the rules described above (line 11). Then, from the resulting sets of execution intervals, Algorithm 3 extracts the execution intervals causing the conflicts in the current RZ which will result in *Comb* (line 12). *Comb* may contain two or more tasks. Each obtained *Comb* is processed individually to study the load of the RZ (line 13–line 24). For each *Comb*, we start the study only at the time at which all the tasks coexist to check for possible conflict and its consequence on the current RZ. Thus, Algorithm 3 searches the latest tasks $\{B\}$ in *Comb* (line 14) and for tasks $\{D\}$ that either have already started their executions and still running after $\{B\}$ arriving or have been ready before $\{B\}$ arriving; it searches their remaining execution times and periods (line 15–line 17) by promoting the tasks with the earliest deadlines and using the ASAP technique, especially in the case there are more than two tasks within the *Comb*. Finally, Algorithm 3 computes the

(1) A, B, D // Tasks
(2) i, j, k, m // Natural, iterations of execution of tasks
(3) *Execution-Interval*$_{A_i}$ // The set of execution intervals for task A during i^{th} iteration
(4) *Crossing-Combination* // All the possible combinations of sets *Execution-Interval*$_{A_i}$ of distinct tasks that give overlapping execution intervals on a given RZ, during HP
(5) *Comb* // Combination of overlapping execution intervals depicted by [A-min, A-max]
(6) C_D^r // The remaining execution time within task D
(7) P_D^r The remaining period within task D
(8) *Pmax* // The period of conflict for a combination of overlapping execution intervals
(9) **for all** iteration i **do**
(10) **for all** RZ **do**
(11) *Crossing-Combination* = $\{\{\text{Execution-Interval}_{A_j}\}/\bigcap_{j\leq i} \text{Execution-Interval}_{A_j} \neq \oslash\}$
(12) *Comb* = one combination of overlapping execution intervals extracted from sets in *Crossing-Combination*.
(13) **for all** *Comb* in *Crossing-Combination* **do**
(14) In *Comb*, search the latest tasks $\{B\}$ in starting execution
(15) In *Comb*, search all the remaining tasks $\{D\}$ that are ready or start execution before $\{B\}$ and determine their remaining execution times: C_D^r and their remaining periods: P_D^r
(16) $C_D^r = C_D - (B\text{-min} - D\text{-min})$
(17) $P_D^r = P_D - (B\text{-min} - D\text{-min})$
(18) **if** ($\forall D$-min, D-min = $Tready_{D_1} + (k-1) * P_D$, $\forall D$-max, D-max = $Tready_{D_1} + k * P_D$, $\forall B$-min, B-min = $Tready_{B_1} + (m-1) * P_B$ and $\forall B$-max, B-max = $Tready_{B_1} + m * P_B / Tready_{D_1}$ $\in \{Treadymin_{D_1}, \ldots, Treadymax_{D_1}\}$, $Tready_{B_1} \in \{Treadymin_{B_1}, \ldots, Treadymax_{B_1}\}$, and k and m are the iterations from which the execution intervals are taken respectively for tasks $\{D\}$ and $\{B\}$) **then**
(19) *RZ-Load* = $\sum_D C_D^r/P_D^r + \sum_B C_B/P_B$
(20) **else**
(21) *Pmax* = $\max_D(D\text{-max} - B\text{-min})$
(22) *RZ-Load* = $\sum_D C_D^r/Pmax + \sum_B C_B/Pmax$
(23) **end if**
(24) **end for**
(25) **end for**
(26) **end for**

Algorithm 3: Search of overlapping execution intervals and RZ loads.

load of the current RZ for this current *Comb* according to two cases. In the first case (Case 1, line 18-line 19), the remaining tasks with their new execution time values and periods are considered as virtual new tasks, and if their execution intervals in *Comb* corresponds to their total periods, Algorithm 3 intuitively computes the load of the RZ as mentioned in line 19 by considering the occupation rate (C_A/P_A) of each virtual new task A and each latest task $\{B\}$ in *Comb* on the current RZ.

In the second case (Case 2), the longest period of time (*Pmax*) that could be shared by the tasks in *Comb* is determined in line 21 and the load of the RZ is studied during *Pmax* (line 22).

This step deals with all possible cases of execution conflicts on all the RZs. At the end of this step, we obtain at each iteration during the HP, the loads of each RZ produced by each combination of tasks giving overlapping execution intervals. Consequently, we can detect the possible overloads in the RZs (*RZ-load* > 1). Some combinations might be included within other ones. When overloads are detected on some RZs, the next step resolves the problem either by performing migration of tasks respecting their preemption points or by incrementing the number of overloaded RZs until the overloads are covered.

(c) Task Migration or Addition of RZs. Migration of tasks causing an overload on a given RZ during a combination of simultaneous executions might be total or partial. Total migration consists in replacing the entire execution of one or many tasks on another RZs. Partial migration consists in the reallocation of some execution sections within tasks predefined by their preemption points to nonoverloaded RZs. The migration is performed as explained by Algorithm 4. The worst-case temporal complexity of Algorithm 4 is $O(2^{\text{NT}*\text{HP}} * \text{NT} * \text{NZ})$. Algorithm 4 searches the combinations producing overloaded RZs obtained by Algorithm 3 (line 6). Firstly, Algorithm 4 starts by total migration (line 8–line 15) to avoid the preemption of tasks resulting in difficulties related to context switches. During each *Comb* causing overload, the algorithm extracts the execution interval of the task that provides the largest occupation rate in the current *Comb* (line 10). In the case of equality between tasks, the task producing the fewest RZs for total migration during the *Comb* should be selected. The algorithm then determines the execution iteration of the task corresponding to this extracted interval and checks whether the iteration is also studied in another nonoverloaded RZs. Should this be the case, total migration of the task is allowed, and the task is eliminated from the overloaded RZ (line 11). If there are several RZs accepting total migration of the selected

```
(1) Comb // Combination of overlapping execution intervals
(2) Non-overload-comb // Boolean controlling after migration whether RZ is no longer overloaded by Comb
(3) RZ-Load // Load of the RZ corresponding to Comb
(4) RZ-migration // The set of RZs helpful for partial migration
(5) Task-migration // The set of tasks that might perform partial migration
(6) for all Comb giving overloaded RZ do
(7)       Non-overload-comb = False
(8)       while (Non-overload-comb = False) and (Comb ≠ ⊘) do
(9)          // Total migration of tasks
(10)         Select the interval from Comb that gives the most heavy occupation rate and discard it from Comb.
(11)         Check whether the iteration, corresponding to the execution interval of the selected task, is studied on another
              non-overload RZ and update the load of the overloaded RZ after the elimination of the selected task.
(12)         if RZ-Load ≤ 1 then
(13)            Non-overload-comb = True
(14)         end if
(15)      end while
(16)   if Non-overload-comb = False then
(17)      Reinitialize RZ-Load and Comb with its tasks
(18)      Task-migration = ⊘
(19)      RZ-migration = ⊘
(20)      // Partial migration of tasks
(21)      Omit the tasks from the overloaded RZ, corresponding to Comb, that are also acceptable by another RZs (D ≠ ∞) and
          reduce their occupation rates from the loads of these RZs. These latter RZs with the overloaded RZ corresponding to
          Comb are included in the set RZ-migration. The omitted tasks are included in Task-migration
(22)      while Task-migration ≠ ⊘ do
(23)         In Task-migration set, start by the task that gives the best trade-off between least number of RZs in RZ-migration
             where it could migrate and heaviest occupation rate in the overloaded RZ.
(24)         During Comb, within the selected task, choose the biggest execution sections that could be placed in RZs from the
              set RZ-migration without overloading them.
(25)         Update the load of RZs receiving execution sections from the selected task
(26)         if Some execution sections of the selected task are not placed then
(27)            Reinitialize the loads of RZs in RZ-migration to values before processing Comb
(28)            Increment the number of RZ corresponding to Comb up to ⌈RZ-Load⌉, go to 6
(29)         else
(30)            Discard the selected task from Task-migration.
(31)         end if
(32)      end while
(33)      // All the execution sections of tasks are placed
(34)      Non-overload-comb = True
(35)   end if
(36) end for
```

Algorithm 4: Total and partial migration or addition of RZs.

task, the RZ which is least required by tasks in the *Comb* is chosen. In cases of equality, the least loaded RZ is kept. After each total migration of a task, Algorithm 4 updates the load of the RZ corresponding to *Comb* and checks whether it is no longer overloaded (line 12–line 14). When total migration fails to resolve the overload in the RZ, partial migration takes place (line 16–line 35). Partial migration reinitializes the load of the current RZ corresponding to *Comb*. It searches the tasks in *Comb* that give finite D with other RZs and omits their occupation rates from the combinations on these latter RZs and from the current overloaded RZ considering the iteration of each task in *Comb* and includes the omitted tasks in the set *Task-migration*. The RZs accepting the tasks of *Comb* and the RZ corresponding to *Comb* are inserted within the set *RZ-migration* (line 21). Algorithm 4 attributes weights to tasks within *Task-migration* according to their occupation rates in *Comb* and according to the numbers of RZs in *RZ-migration* producing finite D with them. The task yielding the best trade-off between the two parameters is selected (line 23). Within the selected task, partial migration tries to reallocate its predefined execution sections in RZs from *RZ-migration* without causing an overload (line 24). Partial migration promotes the biggest execution sections respecting this rule. It starts by placing the selected execution section on the RZ which is least required by the other tasks in *Task-migration* waiting for partial migration. In cases of equality, it starts with the least loaded RZ in *RZ-migration*. Algorithm 4 pursues the processing of these partial migrations until the set *Task-migration* is empty. If partial migration does not successfully map all the execution sections of a given task in *Task-migration* (line 26), Algorithm 4 reinitializes the loads of the RZs to

their initial values before processing the *Comb* (line 27), stops its processing for the current *Comb*, and increments the number of the corresponding RZ to ⌈*RZ-Load*⌉ (line 28). When a given overloaded *Comb* is resolved by partial migration, Algorithm 4 takes into account the update of loads of all altered RZs (line 25) to be considered for the next *Comb*. Although migration might resolve many overloads for several combinations, it is still very difficult to perform as it is exhaustive and depends on the initial choices of tasks and RZs. One could consider the best case of studied migrations to minimize the number of added RZs. After each increment of RZ, the step must consider the added RZs to deal with the overloads of the remaining *Comb* not yet processed to avoid an excessive number of unusable RZs. As the proposed algorithm of migration is not exact, it might lead to an excessive number of RZs. This problem will be covered during resolution of mapping/scheduling which also optimizes the number of used RZs for the purpose of resource efficiency. However, an excess of RZs is very useful as it guarantees elimination of the infeasibility of analytic resolution and consequently, and it guarantees the feasibility of scheduling of the DAG.

At the end of this step, the number of RZs required to perform the scheduling of the chosen DAG on the FPGA is obtained. The resulting RZs constitute the target multi-reconfigurable-unit system where the scheduling of DAG will be conducted. The next step focuses essentially on determining the optimal valid scheduling that respects the predefined constraints.

3.3.4. Mapping and Scheduling Resolution. In the last step of the current stage, we concentrate on the resolution of mapping and scheduling tasks in the DAG on the resulting multi-reconfigurable-unit system. Mapping and scheduling are highly interlinked. It is well known that static multiprocessor scheduling of DAGs is performed by means of two actions: (i) assignment of an execution order expressed by temporal scheduling and (ii) assignment of processors expressed by mapping, for a set of tasks characterized by precedence and real-time constraints. With our methodology, based on a preemptive model, mapping consists in assigning each task to the most suitable RZs in terms of utilization of costly resources. Mapping is considered as spatial scheduling to a limited number of heterogeneous RZs. The scheduling searches the optimal scenario for task execution on RZs during the HP. At each execution iteration for a given task, it assigns for its execution sections specific times to launch their executions on the corresponding RZs. This scheduling is valid only when it satisfies predefined temporal constraints, and it should optimize the makespan of the graph, parallel efficiency, waiting time, and schedule response time. The proposed resolution leads to global static pipelined scheduling on a heterogeneous multi-reconfigurable-unit system because it is constructed at compile time, and it allows overlapping between execution iterations of distinct tasks on distinct RZs. Moreover, the problem of mapping/scheduling is a combinatorial optimization problem as it uses a discrete solution set and chooses the best combination of feasible assignments by optimizing a multiobjective function.

In this paper, resolution of mapping/scheduling is performed by mixed integer nonlinear programming solver as it is well adapted for this kind of problem. The mapping/scheduling problem is modeled by the quadruplet (constants, variables, constraints, and objective function).

Constants

NT:	Number of tasks constituting the DAG
NZ:	Number of RZs resulting from the task migration or addition of RZs analysis
NP:	Number of RB types existing in the target technology
i, o:	The references of iterations of executions during the HP
j:	The references of RZs
A, B:	The references of tasks
k:	The references of RB types
l, e:	The references of preemption points in tasks
t:	The references of times values, $t \in \{0, \ldots, \infty\}$
$D(A, \mathrm{RZ}_j)$:	The cost D between task A and RZ RZ_j
$RBCost_{\mathrm{k}}$:	The cost of each RB type
HP:	The hyperperiod in the DAG
C_A:	The worst case execution time of task A
P_A:	The period of task A which is equal to its relative deadline
$NbrPreemp_A$:	The number of possible preemption points within task A
$Preemp_{A,l}$:	The set of possible preemption points of task A. The first preemption point for all tasks is equal to 0
$Section_{A,l}$:	The execution section within task A provided by the predefined preemption point l
$NbrIter_A$:	The number of execution iterations of task A during the HP
$TimesValues_t$:	The set of possible times assigned to preemption points during the HP. It is equal to $\{0, \ldots, \infty\}$
$Pred_{A,B}$:	Binary constant takes 1 when task A has a precedence constraint with task B in the DAG
$Depend_A$:	Binary constant takes 1 when task A has data dependence constraints with tasks in the DAG
$Config_j$:	The configuration overhead of RZ_j
Com:	The maximum value of time for transmitting a data of unit length between two dependent tasks
$y_{A,B}$:	The amount of data sent by B for A execution.

Variables

$TUnicity_{j,A,l,t,i}$. Binary variable takes 1 when the preemption point l of task A is mapped to RZ_j at time t at iteration i. This

variable ensures the link between mapping and scheduling. In our resolution, the mapping/scheduling problem is solved when binary values are assigned to all these variables.

$PTimeRZ_{j,A,l,i}$. This variable represents the time value assigned to preemption point l of task A on RZ_j at iteration i during the HP. This variable is not defined when RZ_j gives infinite D with task A. It is obtained as expressed by

$$PTimeRZ_{j,A,l,i} = \sum_{\substack{t \\ 0 \le TimeValues_t \le \mathrm{HP}}} TUnicity_{j,A,l,t,i} \times TimeValues_t,$$
$$\forall 1 \le j \le \mathrm{NZ}, \quad 1 \le A \le \mathrm{NT}, \quad 1 \le l \le NbrPreemp_A,$$
$$1 \le i \le NbrIter_A, \quad D\left(A, \mathrm{RZ}_j\right) \neq \infty. \tag{12}$$

$PTime_{A,l,i}$. This variable provides the time value assigned to preemption point l of task A at iteration i during the HP and is calculated by means of

$$PTime_{A,l,i} = \sum_{\substack{1 \le j \le \mathrm{NZ} \\ D(A,\mathrm{RZ}_j) \neq \infty}} PTimeRZ_{j,A,l,i}, \quad \forall 1 \le A \le \mathrm{NT},$$
$$1 \le l \le NbrPreemp_A, \quad 1 \le i \le NbrIter_A. \tag{13}$$

$Occupation_{j,A}$. The total duration of execution of task A on RZ_j, after achievement of mapping/scheduling; it is computed by summing up all the execution sections of A mapped to RZ_j. This variable is not defined when RZ_j gives infinite D with task A. It is obtained using

$$Occupation_{j,A} = \sum_{\substack{l,i \\ 1 \le l \le NbrPreemp_A \\ 1 \le i \le NbrIter_A}} \sum_{\substack{t \\ 0 \le TimeValues_t \le \mathrm{HP}}} \left(TUnicity_{j,A,l,t,i} \times Section_{A,l}\right)$$
$$\forall 1 \le j \le \mathrm{NZ}, \quad 1 \le A \le \mathrm{NT}, \quad D\left(A, \mathrm{RZ}_j\right) \neq \infty. \tag{14}$$

$Exist_{j,A,l,i}$. This Binary variable tests whether the preemption point l of task A during its execution iteration i is mapped to RZ_j. This variable is not defined when RZ_j gives infinite D with task A. It is obtained by means of

$$Exist_{j,A,l,i} = \sum_{\substack{t \\ 0 \le TimeValues_t \le \mathrm{HP}}} TUnicity_{j,A,l,t,i}, \quad \forall 1 \le j \le \mathrm{NZ},$$
$$1 \le A \le \mathrm{NT}, \quad 1 \le l \le NbrPreemp_A,$$
$$1 \le i \le NbrIter_A, \quad D\left(A, \mathrm{RZ}_j\right) \neq \infty. \tag{15}$$

Constraints

Infeasibility of Mapping for Preemption Points. The constraint expressed by (16) prohibits the mapping of preemption points of task A to RZ_j giving infinite D with the task. Indeed, as explained in the second step of the task clustering stage, infinite D between a task and an RZ means that there is a lack of RBs in the RZ preventing task execution or an absence of RB types which are required by the task in the RZ. In addition, this constraint asserts that the time values chosen during mapping/scheduling for preemption points of tasks must be within the set of integers $\{0, \ldots, \mathrm{HP}\}$,

$$TUnicity_{j,A,l,t,i} = 0,$$
$$\text{when } D\left(A, \mathrm{RZ}_j\right) = \infty \quad \text{or} \quad TimeValues_t > \mathrm{HP},$$
$$\forall 1 \le j \le \mathrm{NZ}, \quad 1 \le A \le \mathrm{NT},$$
$$1 \le l \le NbrPreemp_A, \quad 1 \le i \le NbrIter_A, \quad 0 \le t \le \infty. \tag{16}$$

Uniqueness of Mapping/Scheduling Preemption Points on RZs. As expressed by (17), at each possible execution iteration i, if some tasks $\{A\}$ need to be scheduled on RZ_j at a time referred to as t, one task can be executed on this RZ at this time,

$$\sum_{\substack{A,l,i \\ 1 \le A \le \mathrm{NT} \\ 1 \le l \le NbrPreemp_A \\ 1 \le i \le NbrIter_A \\ D(A,\mathrm{RZ}_j) \neq \infty}} TUnicity_{j,A,l,t,i} \le 1, \quad \forall 1 \le j \le \mathrm{NZ},$$
$$0 \le TimeValues_t \le \mathrm{HP}, \quad 0 \le t \le \infty. \tag{17}$$

Uniqueness of RZs for Preemption Points. This constraint asserts that at each execution iteration i, each preemption point l of A must exist on a unique RZ_j (see (18)) and must be scheduled at a unique time referred to as t. This constraint also guarantees the achievement of task execution at each repetition, as all the preemption points delimiting the execution sections of the task are fitted on RZs at specified time values,

$$\sum_{\substack{j,t \\ 1 \le j \le \mathrm{NZ} \\ 0 \le TimeValues_t \le \mathrm{HP} \\ D(A,\mathrm{RZ}_j) \neq \infty}} TUnicity_{j,A,l,t,i} = 1, \quad \forall 1 \le A \le \mathrm{NT},$$
$$1 \le l \le NbrPreemp_A, \quad 1 \le i \le NbrIter_A. \tag{18}$$

Order of Preemption Points of a Task. At each execution iteration i for a task A, the preemption points must be scheduled in order, that is, the time value assigned to a preemption point $l+1$ must be superior to the completion of execution section specified by the preemption point l. Equation (19) guarantees a consistent order of scheduling

of preemption points for a given task and the completion of their corresponding execution sections,

$$PTime_{A,l+1,i} \geq PTime_{A,l,i} + Section_{A,l}, \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq l \leq NbrPreemp_A - 1, \quad 1 \leq i \leq NbrIter_A. \tag{19}$$

Order between Iterations. Based on (19), the constraint expressed by (20) requires that during each execution iteration i for a task A, the scheduling of tasks must lead to the completion of each execution section within A before the beginning of its next iteration $(i + 1)$. Simultaneous executions of distinct iterations for the same task are not permitted,

$$PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A} \leq PTime_{A,1,i+1}, \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A - 1. \tag{20}$$

Upper Bound of Task Execution. Based on (19) and (20), (21) indicates that for each scheduled preemption point for a given task A, at each execution iteration i, its execution section must be completed before the HP. Otherwise, the resulting scenario is not considered a feasible scheduling,

$$PTime_{A,NbrPreemp_A,NbrIter_A} + Section_{A,NbrPreemp_A} \leq \mathrm{HP} \quad \forall 1 \leq A \leq \mathrm{NT}. \tag{21}$$

Nonoverlapping Execution. This constraint eliminates task conflicts on a given RZ. For all execution iterations, when several tasks are scheduled on the same RZ, their predefined execution sections must not overlap (see (22)). This constraint is also applicable within the same task, that is, overlapping running between execution sections is not allowed either during the same iteration, which is implicitly expressed by (19) by the imposed order for preemption point scheduling, or between distinct iterations, which is implicitly expressed by the imposed order for execution iterations of a given task in (20). Equation (22) describes this constraint between two tasks A and B existing on the same RZ_j,

$$Exist_{j,A,l,i} \times Exist_{j,B,e,o} \times \left(PTimeRZ_{j,A,l,i} + Section_{A,l}\right) \leq PTimeRZ_{j,B,e,o},$$

or

$$Exist_{j,A,l,i} \times Exist_{j,B,e,o} \times \left(PTimeRZ_{j,B,e,o} + Section_{B,e}\right) \leq PTimeRZ_{j,A,l,i},$$

$$\forall 1 \leq A, B \leq \mathrm{NT}, \quad 1 \leq j \leq \mathrm{NZ}, \quad 1 \leq l \leq NbrPreemp_A, \quad 1 \leq e \leq NbrPreemp_B, \quad 1 \leq i \leq NbrIter_A, \quad 1 \leq o \leq NbrIter_B. \tag{22}$$

Precedence and Dependence Constraints. Equation (23) defines the precedence and dependence constraints. At each possible execution iteration i of task A, if A has dependence links with tasks $\{B\}$, its execution for this iteration can be launched only if the running within ith execution iterations of all its predecessors $\{B\}$ has been completed and the required data for A execution have been received,

$$PTime_{A,1,i} \geq PTime_{B,NbrPreemp_B,i} + Section_{B,NbrPreemp_B} + Com \times y_{A,B} \quad \forall 1 \leq A, B \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_B, \quad Pred_{B,A} = 1. \tag{23}$$

Periodicity Constraint without Dependence Constraints. This constraint focuses on the periodic execution of tasks without predecessors in the DAG. Each task A is repeated periodically according to its specified period P_A in its functional model. Each repetition defines an execution iteration for the task. Equation (24) imposes constraints only on the first preemption point of the task as (19) defines an order for scheduling the preemption points in the same iteration which consequently will take into account the periodicity constraint for the other remaining preemption points,

$$PTime_{A,1,i} \geq (i-1) \times P_A \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A, \quad Depend_A = 0. \tag{24}$$

Periodicity Constraint with Dependence Constraints. Equation (25) addresses tasks with predecessors in the DAG. These predecessors assert a dependence of execution on their successors at each repetition. Like for tasks without predecessors, tasks having dependence constraints with other tasks in the DAG must be repeated periodically as specified by their functional model and especially taking into consideration the first instants of completion of their predecessors. Indeed, a task with dependence constraints begins its first iteration after the execution achievement of all its predecessors and their data sending. Consequently, the periodicity constraint must consider the beginning time of these tasks with dependence constraints. Similarly, by means of (19), this constraint will be considered for all the preemption points,

$$PTime_{A,1,i} \geq \max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \left(PTime_{B,NbrPreemp_B,1} + Section_{B,NbrPreemp_B} + Com \times y_{A,B}\right) + (i-1) \times P_A, \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A, \quad Depend_A = 1. \tag{25}$$

Deadline Constraint without Dependence Constraints. The key idea behind the constraint explained by (26) is to respect the strict real-time constraints in the case of the absence of

dependence constraints. At each given execution iteration i for a task A without predecessors, the running of task A must be completed before its absolute deadline,

$$PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A} \leq i \times P_A \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A, \quad Depend_A = 0. \tag{26}$$

Deadline Constraint with Dependence Constraints. Equation (27) adheres to the deadline constraint in the case of the existence of dependence constraints between tasks in the DAG. Knowing that the beginning of a given task A with predecessors $\{B\}$ in the DAG is considered since the end of running of all its predecessors within their first execution iterations and the receipt of required data, the absolute deadline of A at each execution iteration i must be met as follows:

$$\begin{aligned} &PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A} \\ &\leq \max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \left(PTime_{B,NbrPreemp_B,1} + Section_{B,NbrPreemp_B} + Com \times y_{A,B}\right) + i \times P_A, \quad \forall 1 \leq A \leq \mathrm{NT}, \\ &\qquad 1 \leq i \leq NbrIter_A, \quad Depend_A = 1. \end{aligned} \tag{27}$$

Objective Function (F). The objective function guides resolution and helps to converge to the optimal solution. The minimization objective function F in (28) defines the optimal solution for the mapping/scheduling sub-problem by means of six parameters. F promotes the solution that provides the best trade-off of lowest values for these parameters. The parameters are all considered important for our methodology and are weighted according to their ranges of values. However, F gives priority to the number of occupied RZs more than other parameters. In fact, F increases exponentially with the minimum growth of the number of used RZs. Preference for this parameter is explained by the fact that our methodology is strongly dependent on physical architecture; hence, the minimum number of RZs enabling scheduling and satisfying the strict predefined constraints must be determined in order to avoid resource wastage and its consequences,

$$\begin{aligned} F &= \ln(Param) + \exp(NumberOfOccupiedRZs), \\ Param &= \delta_1 \times MakeSpan + \delta_2 \times WaitingTime + \delta_3 \times ResourceOptimization + \delta_4 \times MigrationNumber + \delta_5 \times ConfigurationOverhead. \end{aligned} \tag{28}$$

MakeSpan. It is determined by the length of the obtained scheduling. This parameter is considered the most pertinent factor in multiprocessor scheduling of DAGs as it evaluates the efficiency of the performed scheduling in terms of parallelism on processors and execution speed. Equation (29) reduces the *MakeSpan* of the DAG by minimizing the time of finishing execution within the last execution iterations for all tasks as they are linked by precedence and dependence constraints. Consequently, as the execution iterations are also highly dependent, it is necessary to start the execution of a given task for each iteration as soon as possible in order to minimize the makespan of the DAG,

$$MakeSpan = \max_{\substack{A \\ 1 \leq A \leq \mathrm{NT}}} \left(PTime_{A,NbrPreemp_A,NbrIter_A} + Section_{A,NbrPreemp_A}\right). \tag{29}$$

WaitingTime. It is determined by the response time of the scheduling for task execution. By means of (30), *WaitingTime* is computed as the sum of differences between the obtained runtimes of the tasks during the scheduling span and their effective execution times. *WaitingTime* includes the waiting time of tasks in the ready queue of the scheduler waiting for execution when their dependence and periodicity constraints are satisfied and the blocking time when the task is preempted,

$$\begin{aligned} &WaitingTime = \sum_{\substack{A \\ 1 \leq A \leq \mathrm{NT}}} (RunTime_A - NbrIter_A \times C_A), \\ &RunTime_A = \sum_{\substack{i \\ 1 \leq i \leq NbrIter_A}} \left(\left(PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A}\right) - (i-1) \times P_A\right) \\ &\text{if } Depend_A = 0, \\ &RunTime_A = \sum_{\substack{i \\ 1 \leq i \leq NbrIter_A}} \left(\left(PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A}\right) - \max\left(\max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \left(PTime_{B,NbrPreemp_B,1} + Section_{B,NbrPreemp_B} + Com \times y_{A,B}\right) + (i-1) \times P_A, \right.\right. \\ &\qquad \left.\left. \max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \left(PTime_{B,NbrPreemp_B,i} + Section_{B,NbrPreemp_B} + Com \times y_{A,B}\right) \right) \right), \\ &\text{else} \end{aligned} \tag{30}$$

ResourceOptimization. It is related to the physical features of the chosen technology. The parameter considers resource wastage and the utilization of costly resources. Both issues are involved in cost D computation in the first stage. Thus, resource optimization, using (31), targets mapping tasks with high occupation rates to the RZs providing the lowest cost D with them,

$$\begin{aligned} &ResourceOptimization \\ &= \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}}} \left(\frac{D\left(A,\ \mathrm{RZ}_j\right)^2 \times Occupation_{j,A}{}^2}{4} \right. \\ &\qquad \left. - D\left(A, \mathrm{RZ}_j\right) \times Occupation_{j,A} \right). \end{aligned} \tag{31}$$

MigrationNumber. Although the migration concept optimizes scheduling length, it helps to guarantee real-time constraints and to increase resource efficiency; it also raises configuration overhead. Bearing this in mind, we constructed (32) that aims at minimizing the number of migrations required. For a given task A, the first expression of (32), on the left of the summation, searches the number of migrations within the same execution iteration, and the second part of the summation calculates the performed migrations between iterations,

$$\begin{aligned} MigrationNumber = &\sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i\le NbrIter_A}} \ \sum_{\substack{1\le l<NbrPreemp_A\\ Exist_{j,A,l,i}\neq 0,\ Exist_{j,A,l+1,i}=0}} 1 \\ &+ \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}}} \ \sum_{\substack{1\le i<NbrIter_A\\ Exist_{j,A,NbrPreemp_A,i}\neq 0,\ Exist_{j,A,1,i+1}=0}} 1. \end{aligned} \tag{32}$$

ConfigurationOverhead. During the scheduling span, we do not consider the configuration overhead before the task execution on the RZ. This configuration overhead is negligible, that is, according to several experimentations performed with Virtex 5 technology, which uses parallel high-speed configuration ports, it represents less than 10% of the computation time of the task and does not impact the real-time functioning. However, after scheduling has been achieved, the configuration overhead, that impacts scheduling performance and power consumption, is evaluated using (33). The configuration overhead parameter should be reduced during resolution of mapping/scheduling. Equation (33) includes four expressions in the summation. The first expression, *Exp*1*,* is the initial configuration when tasks are launched by the scheduler for their first execution iteration as they did not exist on the RZs. The second expression, *Exp*2*,* represents the configuration overhead required to configure a task that migrates from one RZ to another within the same iteration i; the third expression, *Exp*3*,* depicts the configuration overhead required by a task that has finished its execution on a given RZ for an iteration i and started its $(i+1)$th iteration by migrating to another RZ. The fourth expression, *Exp*4, computes the configuration overhead resulting from intermediate tasks preempting a given task running on the same RZ. The first part of *Exp*4 considers the configuration overhead required in cases where a task A, during iteration i, finishes its execution section delimited by preemption point l, then it is preempted by other tasks to perform execution sections, which is then followed by task A performing its $l+1$ execution section at a later stage on the same RZ. This event is detected by the binary variable $CurrentIter_{j,A,l,i}$. This binary variable takes 1 if two successive execution sections for the same task at the same iteration are separated by one or more tasks on the same RZ. By means of variable $InterIter_{j,A,NbrPreempA,i}$, the second part of *Exp*4 deals with situations during which a task finishes its last execution section for a given iteration i on an RZ and starts the first execution section of $(i+1)$th iteration on the same RZ after the execution of other execution sections of other tasks on the same RZ. This variable takes 1 when at least one other task runs on the same RZ between two successive execution iterations of the same task,

$$\begin{aligned} &ConfigurationOverhead = Exp1 + Exp2 + Exp3 + Exp4 \\ &Exp1 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 0\le TimeValues_t\le \mathrm{HP}}} TUnicity_{j,A,1,t,1} \times Config_j \\ &Exp2 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i\le NbrIter_A}} \ \sum_{\substack{1\le l<NbrPreemp_A\\ Exist_{j,A,l,i}\neq 0,\ Exist_{j,A,l+1,i}=0}} Config_j \\ &Exp3 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}}} \ \sum_{\substack{1\le i<NbrIter_A\\ Exist_{j,A,NbrPreempA,i}\neq 0,\ Exist_{j,A,1,i+1}=0}} Config_j \\ &Exp4 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i\le NbrIter_A\\ 1\le l<NbrPreemp_A}} Config_j \times CurrentIter_{j,A,l,i} \\ &\qquad + \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i<NbrIter_A}} Config_j \times InterIter_{j,A,NbrPreemp_A,i}. \end{aligned} \tag{33}$$

NumberOfOccupiedRZs. As explained in the previous step (determination of task execution intervals and the number of RZs), the obtained number of RZs ensures the feasibility of scheduling but is not necessarily the optimal number for valid scheduling. Thus, for the purpose of resource efficiency, (34) searches the lowest possible number of RZs required for

feasible real-time scheduling for the chosen DAG on a multi-reconfigurable-unit system,

$$NumberOfOccupiedRZs = \sum_{\substack{1 \le j \le \mathrm{NZ} \\ RZOccupation_j \neq 0}} 1,$$
$$RZOccupation_j = \sum_{\substack{A \\ 1 \le A \le \mathrm{NT}}} Occupation_{j,A}. \quad (34)$$

Other optimization parameters are also integrated in objective function F, but they are not mentioned in the formulation above as they implicitly impact the makespan. These parameters optimize the solutions that launch the execution of tasks as soon as their predecessors terminate their computations. In addition, they aim to bring the beginning of the tasks closer to the start time of each repetition.

After resolution of the first sub-problem of mapping/scheduling tasks on RZs, the resulting RZs must be placed on the target device. The second sub-problem of task cluster placement is a well-known problem in reconfigurable computing systems, and it differentiates our problem of scheduling on reconfigurable devices from that on software multiprocessors, generally not constrained by the physical architecture of the homogeneous processors. The second sub-problem of RZ placement is described in the next and last stage of our proposed methodology and like mapping/scheduling; it is solved by means of mixed integer programming as it is considered to be a combinatorial optimization problem.

3.4. Partitioning of Reconfigurable Device and Fitting of RZs. In general, the problem of hardware task placement includes two primary subfunctions: (i) *partitioning*, which handles free resource space on the reconfigurable device to identify potential sites enabling hardware task execution and (ii) *fitting*, which, according to the chosen criteria, selects a feasible placement solution among sites provided by the partitioning to fit the hardware task on a suitable physical location. Several research works have been proposed for the placement of hardware tasks on reconfigurable devices, and placement can be divided into on-line and off-line placement. For on-line placement, most scenarios propose as a partitioning technique to search for the maximal empty physical rectangles in the free space to avoid the resource wastage. For example, the two techniques proposed by Bazargan et al. in [13], one referred to as "*Keeping All Maximal Empty Rectangles*" searches all overlapping maximal empty rectangles and the other referred to as "*Keeping Nonoverlapping Empty Rectangles*" keeps only the distinct holes in the free space. In [14], Walder et al. describe efficient partitioning algorithms, the main one being the *On-The-Fly* (*OTF*) partitioner which resizes empty rectangles only if the new arrived task overlaps them. Handa and Vemuri introduce staircase partitioning in [15]. Marconi et al. extend in [16] Bazargan's partitioner by means of an *Intelligent Merging* (*IM*) algorithm that dynamically combines three techniques for managing free resources. In [17], Ahmadinia et al. present a new method of on-line partitioning which manages occupied space on devices rather than free space, given the difficulty of managing empty space and the resulting vast increase in empty rectangles. Regarding the fitting subfunction, the works are essentially based on bin-packing rules, such as in [13] which describes the *First Fit*, *Best Fit*, and *Bottom left* bin-packing algorithms. In [14], Walder et al. employ a hash matrix to perform fitting based on *Best Fit*, *Worst Fit*, *Best Fit with Exact Fit*, and *Worst Fit with Exact Fit* algorithms. In [17], Ahmadinia et al. fit tasks on sites, reducing communication costs, by means of the *Nearest Possible Position* algorithm. For off-line placement, many research works deal with metaheuristics such as simulated annealing, greedy research, and *Keeping All Maximal Empty Rectangle-Best Fit* as described in [13]. In [18], ElGindy et al. describe task rearrangement by using a genetic algorithm approach with the *First Fit* strategy. Considering task placement as a 2D packing problem, researchers propose off-line heuristics such as Lodi et al. who describe the *Next-Fit Decreasing Height*, *First-Fit Decreasing Height*, and *Best-Fit Decreasing height* algorithms in [19], as well as the *Floor-Ceiling* and *Knapsack packing* algorithms in [20]. Integer linear programming is also proposed by Panainte et al. in [21] to model the problem of hardware task placement by minimizing the resources area that is reconfigured at runtime.

With our methodology, we focus on off-line placement of RZs on reconfigurable devices. As the reconfigurable devices afford a limited number of reconfigurable resources and DAG includes bounded numbers of tasks, analytic resolution of the placement sub-problem is the recommended method as it guarantees the optimal solution to perform efficient allocation of tasks on FPGA. The placement of RZs is a combinatorial mono-objective problem. It consists in searching for suitable coordinates for each RZ in the FPGA among all the possible combinations of discrete coordinates. In this section, similarly to the mapping/scheduling sub-problem, the resolution of RZ placement on the reconfigurable device is accomplished through mixed integer programming and optimizes resource efficiency. The achievement of RZ placement results in a 2D physical locations for each task cluster. These physical locations are depicted by reconfigurable physical blocs (RPBs) and are represented by their RB model as described in (35). RZs are abstractions of hardware task clusters, and RPBs are the final reconfigurable units where tasks may be executed. With the concept of run-time partial reconfiguration, after the mapping/scheduling results are obtained respecting the strict predefined constraints and optimizing several criteria, task execution sections are reconfigured and executed on RPBs as restricted by the schedule scenario. Thus, the task bitstreams on their associated RPBs are created at compile time. These bitstreams are used during the DAG running as indicated by scheduling which specifies the corresponding state for each task on each RPB,

$$\mathrm{RPB}_j\_\mathrm{RB} = \left\{\gamma_{j,k}\mathrm{RB}_k\right\}, \quad \gamma_{j,k} \in \mathbb{N},$$
$$1 \le j \le \mathrm{NZ}, \quad 1 \le k \le \mathrm{NP}. \quad (35)$$

As shown in Figure 7, the partitioned RPBs on a reconfigurable device for a given RZ must include all its required RB types to ensure the execution of the tasks. The partitioned RPBs for a given RZ might contain some RBs that are not required by the RZ. For instance, RB_4 is inserted within RPB_3 but is not used by the corresponding RZ. This resource inefficiency is explained by the rectangular shape of the RPBs. Thus, the number of RBs included in the RPB could exceed the required RBs in the RZ. Resource inefficiency is also due to the heterogeneity of the device; partitioning could book some RBs which are not used by the RZ. RB utilization efficiency is an important metric parameter for evaluating placement quality. During RZ placement resolution, we focus on fitting RZs as closely as possible to RPBs in terms of number and types of RBs.

By means of mixed integer linear programming solver, both the partitioning and fitting subfunctions are solved simultaneously. The RZ placement sub-problem is modeled by the quadruplets (Constants, Variables, Constraints, and Objective Function).

Constants

NZ:	Number of RZs resulting from mapping/scheduling resolution
NP:	Number of RB types existing in the target technology RZ features, RB features
Device features	
Device_Width:	The width of the device
Device_Height:	The height of the device
Device_RB:	The RB model of the device.

Variables

Placement resolution consists in assigning discrete values for the four coordinates specifying the RPB_j for each RZ_j. Each RPB_j is constructed by four coordinates:

X_j:	The abscissa of the upper left vertex of RPB_j
Y_j:	The ordinate of the upper left vertex of RPB_j
$WRPB_j$:	The abscissa of the upper right vertex of RPB_j
$HRPB_j$:	The ordinate of the bottom left vertex of RPB_j.

Constraints

Heterogeneity Constraint. As RZs are fitted on RPBs, during RPB partitioning, the number of each RB type within the RPBs (X_j, $WRPB_j$, Y_j, $HRPB_j$) must be greater than or equal to those in the RZs ($\beta_{j,k}$) as formulated by (36) in order to satisfy the RB requirements of the RZs. Because of the heterogeneity of RBs in the device and the rectangular shape of RPBs, the partitioned RPBs could include some RB types not required by the RZs. Moreover, the number of RB types in the RPBs and included in the RZs might exceed that required by the RZs. This resource inefficiency is minimized by means of the objective function,

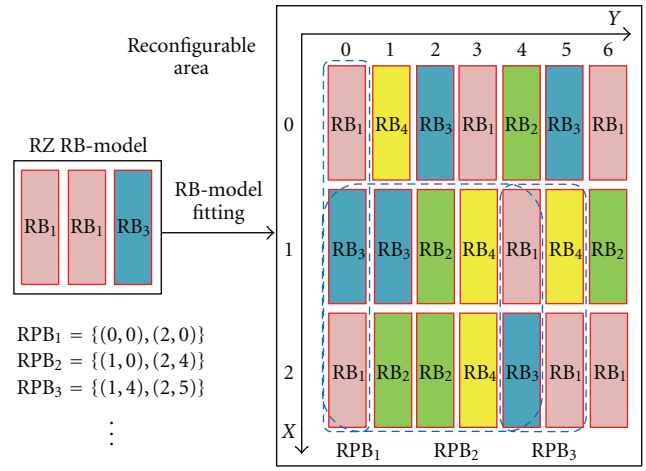


Figure 7: Example of RPBs for RZ.

$$\beta_{j,k} \leq \sum_{\substack{X_j \leq m \leq WRPB_j \\ Y_j \leq n \leq HRPB_j}} \sum_{device\_RB[m][n]=\mathrm{RB}_k} 1,$$

$$\forall 1 \leq j \leq \mathrm{NZ}, \quad 1 \leq k \leq \mathrm{NP},$$

$$\mathrm{RZ}_j\_\mathrm{RB} = \left\{\beta_{j,k}\ \mathrm{RB}_k\right\}, \quad \mathrm{RPB}_j\left(X_j, WRPB_j, Y_j, HRPB_j\right). \tag{36}$$

Nonoverlapping between RPBs. As expressed by (37), this constraint restricts the fitting of RZs on nonoverlapping RPBs

$$\begin{aligned} &X_q > WRPB_j \quad \text{or} \quad X_j > WRPB_q \\ &\text{or} \quad Y_q > HRPB_j \quad \text{or} \quad Y_j > HRPB_q, \\ &\forall j \neq q, \quad 1 \leq j, \quad q \leq \mathrm{NZ}. \end{aligned} \tag{37}$$

Objective Function (F). Objective function F comprises one parameter which is *ResourceEfficiency*. This primordial parameter focuses on finding the closest RPB partitioned on the FPGA to fit each RZ in terms of number and types of RBs. By respecting both previous constraints, (38) assesses placement quality after RZ insertion on the selected RPBs in order to achieve the optimal solution. Increasing resource efficiency reduces configuration overhead,

$$\begin{aligned} &ResourceEfficiency = \sum_{\substack{1 \leq j \leq \mathrm{NZ} \\ 1 \leq k \leq \mathrm{NP}}} RBCost_k \times \left(\gamma_{j,k} - \beta_{j,k}\right), \\ &\mathrm{RPB}_{j_{\mathrm{RB}}} = \left\{\gamma_{j,k}\ \mathrm{RB}_k\right\}, \quad \mathrm{RZ}_j\_\mathrm{RB} = \left\{\beta_{j,k}\ \mathrm{RB}_k\right\}, \\ &1 \leq j \leq \mathrm{NZ}, \quad 1 \leq k \leq \mathrm{NP}. \end{aligned} \tag{38}$$

Both subproblems, that is, (i) mapping/scheduling tasks on RZs and (ii) partitioning/fitting RZs on the FPGA, are successively solved by means of powerful solvers dedicated

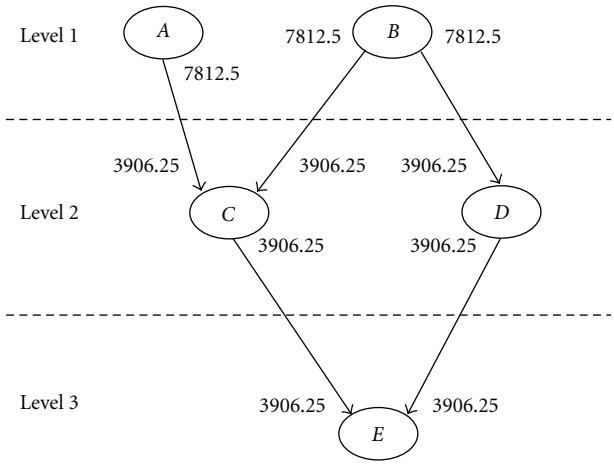


Figure 8: Tasks in the DAG.

by the AIMMS [22] environment. The chosen solvers for mixed integer programming satisfy predefined constraints and provide an optimal solution in an acceptable time frame. The following section shows how our proposed methodology may be applied to an example of DAG on Virtex 5 technology; the section also evaluates scheduling and placement quality.

4. Application and Results

The experiments were separated into two parts. The first part focuses on constructing the DAG as restricted by the rules of the mapping/scheduling stage, detailing the performed analysis and showing the results of task mapping/scheduling on the resulting RZs with a performance evaluation. The second part demonstrates placement resolution of the RZs on the reconfigurable device, Virtex 5 FX70T (ML507 board), and describes the metrics used to measure placement quality.

To illustrate the methodology proposed for static scheduling of DAGs on multi-reconfigurable-unit systems with precedence and deadline constraints, we designed the fully connected DAG shown in Figure 8. Tasks were selected from opencores (http://www.opencores.org/).

Each task is characterized by its worst-case execution time (WCET) which also integrates the communication latency for data sending to task successors, its period (relative deadline), and the set of preemption points. The number of needed slices is also synthesized for each task. In the slices, either only LUTs are used or only registers are used or both of them are used. The number of slices used by the task for LUTs and those for registers are mentioned in the third column of Table 2.

The configuration overheads were computed by (3) and by using our own IP based on the ICAP reconfiguration port having a width of 32 bits and a frequency of 100 MHz. Values on the DAG edges represent the average number of packets of 16 words of 32 bits for communication between tasks.

With the Virtex 5 technology [23], there are four main resource types, that is, CLBL, CLBM, BRAM, and DSP. Considering reconfiguration granularity, the RBs in Virtex 5 are vertical stacks composed of the same type of resources: RB_1 (20 CLBMs), RB_2 (20 CLBLs), RB_3 (4 BRAMs), and RB_4 (8 DSPs). In our application, for Virtex 5 FX70T, we considered that the lower the number of the RB types on the device and the higher its functioning speed, the more its cost would increase. We, respectively, assigned 16, 10, 168, and 194 as the *RBCost* for RB_1, RB_2, RB_3, and RB_4. We modeled the Virtex 5 FPGA and the hardware tasks with their RB-models. The task features are shown in Table 2.

The overview functioning of selected tasks is described below.

Reed-Solomon (*A*). It is an error correcting code that works by oversampling the Galouee's field polynomial constructed from the data to be coded. It is widely used to recover data from possible errors that occur during disk reading.

AES (*B*). Advanced encryption standard can decrypt input data using 256-bit key.

IIR (*C*). It performs infinite impulse response low-pass filter which cut off frequency in the range of 0.1 to 0.4 of the sampling frequency.

FFT (*D*). It performs 64 points Fast Fourier Transform where the data and coefficients are adjustable in the range 8 to 16 bits.

Basic DES (*E*). It performs single Data Encryption Standard by processing 16 identical rounds. Each round encrypts 32-bit block using 64-bit key to provide 32-bit output block.

We applied the proposed methodology for placement and scheduling of DAGs on reconfigurable devices and obtained the following results.

4.1. Task Clustering Results. This stage executes Algorithm 1 which results in RZ types (RZ_1, RZ_2, RZ_3) represented by their RB models in the first column of Table 3. RZ_1 is inserted by *A* and *B*, RZ_2 is provided by *C* and *D*, and task *E* creates RZ_3.

When the maximal numbers of RBs within a constructed RZ are provided by several tasks, the configuration overhead of the RZ must be recomputed as described by (3). In our application, the maximum numbers of RBs within RZs are created by the same task. Thus, the RZ configuration overheads are provided by the predefined task features shown in Table 2. The second step of this stage computes the costs *D* between the DAG tasks and the resulting RZs. In Table 3, the bold numbers are the lowest costs *D* for tasks with the most suitable RZs.

The resolution of scheduling of the chosen DAG on Virtex 5 FX70T is detailed in Sections 4.2 and 4.3.

4.2. Mapping/Scheduling Results and Scheduling Quality Evaluation. DAG behavior is studied within the time interval (HP) of a period equal to the least common multiple of the

Table 2: Task features.

Reference	Name	Slices (LUTs/registers)	WCET (μs)	Period (ms)	Configuration overhead (μs)	Preemption points (μs)	RB model
A	Reed-Solomon	2234/1224	26576	500	1116	14770, 20200	$\{8RB_1, 7RB_2, 1RB_3, 1RB_4\}$
B	AES	1150/507	42733	500	675	5000, 11000, 23000, 30000	$\{5RB_1, 5RB_2, 1RB_3, 1RB_4\}$
C	IIR	678/565	11805	250	524	3334, 10000	$\{4RB_1, 1RB_2, 0RB_3, 1RB_4\}$
D	FFT	2333/2010	11806	250	1199	5000, 11667	$\{9RB_1, 7RB_2, 0RB_3, 1RB_4\}$
E	Basic DES	387/192	22280	250	188	7500, 15000, 19000	$\{2RB_1, 2RB_2, 0RB_3, 0RB_4\}$

Table 3: RZ types and D costs.

	A	B	C	D	E
$RZ_1\{8RB_1, 7RB_2, 1RB_3, 1RB_4\}$	**0**	**68**	292	∞	448
$RZ_2\{9RB_1, 7RB_2, 0RB_3, 1RB_4\}$	∞	∞	**140**	**0**	356
$RZ_3\{2RB_1, 2RB_2, 0RB_3, 0RB_4\}$	∞	∞	∞	∞	**0**

periods of its tasks which is equal to 500 ms. Consequently, the execution iterations of tasks during the HP are deducted as follows:

$$\begin{aligned} NbrIter_A &= \left\lfloor \frac{\mathrm{HP}}{P_A} \right\rfloor = \left\lfloor \frac{500}{500} \right\rfloor = 1, \\ NbrIter_B &= 1, \qquad NbrIter_C = 2, \\ NbrIter_D &= 2, \qquad NbrIter_E = 2. \end{aligned} \tag{39}$$

The three steps preceding mapping/scheduling resolution are detailed hereunder. In the remaining sections of the paper, the values are expressed in μs.

4.2.1. Checking of Precedence, Dependence, and Real-Time Rightness in the DAG. The constraints for checking precedence, dependence, and real-time rightness in the proposed DAG are detailed below.

Dependence Checking. In Figure 8, it is obvious that the periods of tasks between the levels are guarded or dubbed. Thus, the successors are more repetitive than the predecessors.

In this case, all the data produced on an edge linking a source task and its successor must be sufficient and consumed by the latter task. As can be seen in Figure 8, the constraint expressed by (7) is satisfied for all the edges. For example, task B produces at its unique execution iteration a data of size $x_{B,C}$ equal to 7812.5 packets on its outgoing edge. These data are consumed totally by its first successor C during its two iterations where it consumes $y_{C,B}$ (3906.25 packets) data at each iteration. A similar remark can be made for data interchanged on the edge linking B and D.

Precedence Checking. This constraint is satisfied by all the interdependent tasks in the DAG and is checked by means of (8) and (9). $Tready_A = 0$, $Tready_B = 0$, $Tready_C = \max(C_A, C_B) = 42733$, $Tready_D = C_B = 42733$, $Tready_E = \max(Tready_C + C_C, Tready_D + C_D) = \max(42733 + 11805, 42733 + 11806) = 54539$.

Task C and Task B Precedence:

$$K = 1\text{: } 42733 + 250000 = 292733 > 0. \tag{40}$$

Task C and Task A Precedence:

$$K = 1\text{: } 42733 + 250000 = 292733 > 0. \tag{41}$$

Task D and Task B Precedence:

$$K = 1\text{: } 42733 + 250000 = 292733 > 0. \tag{42}$$

Task C and Task E Precedence:

$$\begin{aligned} &K = 1\text{: } 54539 + 250000 = 304539 > 42733, \\ &K = 2\text{: } 54539 + 2 * 250000 \\ &\qquad = 554539 > 42733 + 250000 \\ &\qquad = 292733. \end{aligned} \tag{43}$$

Task D and Task E Precedence:

$$\begin{aligned} &K = 1\text{: } 54539 + 250000 = 304539 > 42733, \\ &K = 2\text{: } 54539 + 2 * 250000 \\ &\qquad = 554539 > 42733 + 250000 \\ &\qquad = 292733. \end{aligned} \tag{44}$$

The above tests prove that each execution iteration A_i of a given task A includes only execution iteration B_i or iterations preceding B_i of each successor B. Consequently, the precedences between tasks in the chosen DAG are correct.

Real-Time Checking. For the purpose of real-time functioning, (10) considers dependence constraints between tasks and makes it possible to verify, in the best case of spatial conditions and by providing an RZ for each task, the rightness of real-time functioning according to computation times and periods assigned to tasks in the DAG.

An example of testing real-time constraints for task C is shown hereunder.

Task C

$$\Pi_C = \{A, B\},$$
$$K = 0:\ \max(0 + 26576, 42733 + 0) + 11805 = 54538 \le \min(42733 + 250000, 500000) = 292733\ (A),$$
$$\max(0 + 42733, 42733 + 0) + 11805 = 54538 \le \min(42733 + 250000, 500000) = 292733(B)$$
$$K = 1:\ (42733 + 250000) + 11805 = 304538 \le \min(42733 + 2 * 250000, 500000) = 500000. \quad (45)$$

By providing an RZ for each task, the deadlines of the tasks are met taking into consideration the dependence and precedence constraints.

The above three tests validate the chosen graph in terms of dependence, precedence, and real-time constraints and in the following section; spatial/temporal analyses are performed in order to determine the required number of RZs allowing valid scheduling for the DAG.

4.2.2. Determination of Lists of Ready Times. The first temporal analysis uses (11) to search the ready times of tasks which are helpful to limit execution intervals. Without considering the number of available reconfigurable units and taking into account precedence, dependence, periodicity, and real-time constraints, the ready times are determined as shown in Figure 9.

4.2.3. Determination of Task Execution Intervals and the Number of RZs. The first step of this spatial/temporal analysis uses Algorithm 2 to determine the possible execution intervals of tasks based on lists of ready times and respecting specified constraints. The possible execution intervals for tasks in the DAG are shown below

$$\textit{Execution-Interval}_{A_1} = \{[0, 215915], [1, 215915], \ldots, [i, 215915], \ldots, [189339, 215915]\},$$
$$\textit{Execution-Interval}_{B_1} = \{[0, 215914], [1, 215914], \ldots, [i, 215914], \ldots [173181, 215914]\},$$
$$\textit{Execution-Interval}_{C_1} = \{[42733, 227720], [42734, 227720], \ldots, [215915, 227720]\},$$
$$\textit{Execution-Interval}_{C_2} = \{[292733, 477720], [292734, 477720], \ldots, [465915, 477720]\},$$
$$\textit{Execution-Interval}_{D_1} = \{[42733, 227720], [42734, 227720], \ldots, [215914, 227720]\},$$
$$\textit{Execution-Interval}_{D_2} = \{[292733, 477720], [292734, 477720], \ldots, [465914, 477720]\},$$
$$\textit{Execution-Interval}_{E_1} = \{[54539, 304539], [54540, 304540], \ldots, [i, i + P_E], \ldots, [227720, 477720]\},$$
$$\textit{Execution-Interval}_{E_2} = \{[304539, \mathrm{HP}], [304540, \mathrm{HP}], \ldots, [477720, \mathrm{HP}]\}. \quad (46)$$

Using Algorithm 3, the analysis therefore integrates the spatial aspect and studies possible conflicts between tasks in shared RZs to detect possible overloads. Algorithm 3 must respect the rules explained in Section 3.3.3(b) to search the possible overlapping execution intervals. At each iteration, for each RZ, Algorithm 3 searches all the combinations of tasks causing overlapping execution intervals and inserts them in *Crossing-Combination* and computes the loads of RZs according to two cases as detailed as follows.

Iteration 1

RZ_1. *Crossing-Combination* = {{*Execution-Interval*$_{A_1}$, *Execution-Interval*$_{B_1}$}}. This *Crossing-Combination* contains several combinations of overlapping execution intervals between A and B. In the worst case of overlapping, such as the overlapping between $[0, 215915]$ from *Execution-Interval*$_{A_1}$ and $[0, 215914]$ from *Execution-Interval*$_{B_1}$, the load of RZ_1 obtained by Case 2 is 32% satisfying the predefined constraints.

RZ_2. *Crossing-Combination* = {{*Execution-Interval*$_{C_1}$, *Execution-Interval*$_{D_1}$}}. Similarly, this *Crossing-Combination* provides several combinations of overlapping execution intervals between C and D. In the worst case of overlapping, such as when both tasks have confused execution intervals equal to $[42733, 227720]$, the load of RZ_2 obtained in Case 2 is 12%. Another possible solution consists in total migration, as expressed by Algorithm 4, of task C to the RZ RZ_1 to improve execution parallelism and to minimize

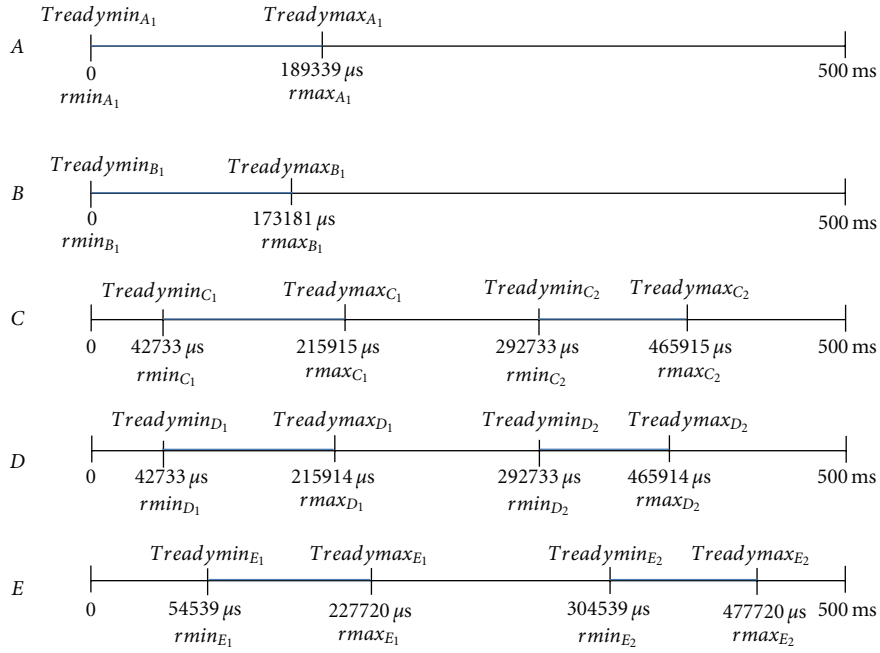


Figure 9: *Treadymin* and *Treadymax* of tasks in the DAG.

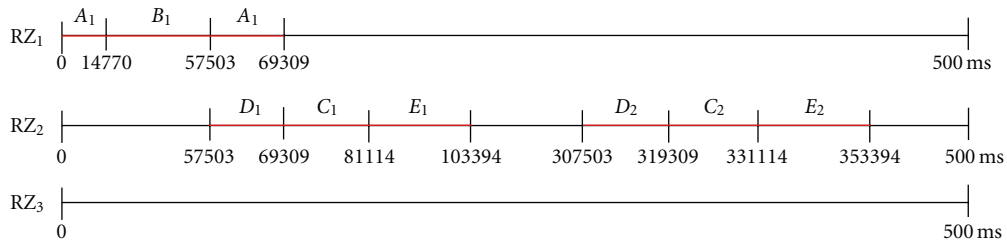


Figure 10: Mapping/scheduling resolution.

the makespan of the DAG. Thus, as soon as tasks A, B computations are achieved, tasks C and D may be launched, respectively, on RZ_1 and RZ_2.

During the first iteration, no combination of detected overlapping execution intervals causes an overload on RZs.

Iteration 2

RZ_2. *Crossing-Combination* = {{*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$}, {*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$, *Execution-Interval*$_{E_1}$}, {*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{D_2}$}, {*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{C_2}$}} {*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$}: As with the first iteration, all the combinations of overlapping execution intervals between tasks C and D do not cause an overload on RZ_2. The load of RZ_2, in the worst case of confused execution intervals, is equal to 12%. Hence, in the worst case, tasks C and D could be executed consequently on RZ_2 respecting all the predefined constraints.

As expressed by the rules set out in Section 3.3.3(b), detection of conflicting tasks must consider current and preceding iterations.

{*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$, *Execution-Interval*$_{E_1}$}: This set provides many combinations of overlapping execution intervals between tasks C, D, and E. However, following Algorithm 3 and the optimization parameters for reducing the makespan as explained at the end of mapping/scheduling resolution, task E starts its first iteration immediately after the achievement of its predecessors C and D which is equal, in the worst case, to 227720 μs, and RZ_2 is idle at this time. Consequent-ly, according to this start time of the first repetition, task E finishes its execution at 250 ms which is not attainable by the least start time of the second iterations of C and D, equal to 292733 μs. Thus, this set leads to no overlapping of execution between C, D, and E and becomes equivalent to the *Crossing-Combination* set: {*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$}. Another possible solution for this set is total migration of C to RZ_1 and E to RZ_3. This solution guarantees efficient parallelism between task computations.

{*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{D_2}$}, {*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{C_2}$}: As explained for the previous set, these sets of overlapping execution intervals do not cause an overload on RZ_2.

RZ_1

Crossing-Combination = {{*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{C_2}$}}. As described for RZ_2 with the last three sets, tasks *C* and *E* do not cause overloads on RZ_1 during this *Crossing-Combination*. In fact, application of Algorithm 3 and the optimization parameters results in no remaining execution of the first repetition of *E* when *C* starts its second execution iteration.

After these spatial and temporal analyses, one can conclude there is no need to perform migration or to add other RZs. The three RZs resulting from the clustering stage are sufficient to perform valid scheduling of the chosen DAG. Thus, RZ_1, RZ_2, and RZ_3 represent the multi-reconfigurable-unit system for DAG scheduling. Based on powerful solvers dedicated by the AIMMS environment, we obtained the following mapping/scheduling which is evaluated in the next section.

4.2.4. Mapping and Scheduling Resolution. Figure 10 describes the mapping/scheduling obtained for five tasks in the DAG on three RZs. As the number of RZs is the highest priority parameter in the objective function, the resolution concludes that for scheduling this DAG, only two RZs are required. RZ_3 is not used, and hence it will not be placed on Virtex 5 FX70T in the third stage. The elimination of RZ_3 enhances resource efficiency and enables the DAG to be extended and performed on the FPGA for future needs. RZ_1 and RZ_2 are selected by the solver to remain in the multi-reconfigurable-unit system since *A* and *B* can execute only on RZ_1, task *D* can execute only on RZ_2, and tasks *C* and *E* can run on both RZs.

The obtained mapping/scheduling takes into account all the optimization parameters described in the sub-problem formulation and satisfies the specified constraints. Tasks *A* and *B* launch their computations first as they have no predecessors, and both can only execute on RZ_1. To reduce the makespan of the DAG, the scheduler decides the preemption of *A* at its second preemption point (14770 μs) to enable the execution of task *B*, the termination of which permits the execution of task *D* on RZ_2. The scheduler decides the execution of *A* before *B* in order to reduce the span between the executions of dependent tasks expressed by the dependence between *C*, *D*, and *E*. Hence, the scheduler promotes the solution that starts *C* immediately after completion of *A* and begins the execution of *E* once its predecessors *C* and *D* complete execution as reinforced by the optimization parameters explained at the end of mapping/scheduling resolution. These parameters aim at bringing closer the execution start of each task to the completion of its predecessors. *C* and *E* can execute on RZ_1 and RZ_2. The mapping assigns their executions to RZ_2 as this RZ provides the best utilization of costly resources guided by their costs *D*. For the purpose of fully exploiting the multi-reconfigurable-unit system and to increase parallel efficiency, task *A* resumes its execution simultaneously with the start time of the first repetition of task *D* at 57503 μs. In their second execution iterations, respecting the periodicity and precedence constraints tasks *C*, *D*, and *E* are performed on their optimal RZ, that is, RZ_2, to optimize resource utilization.

The resulting scheduling respects all the predefined constraints and during task running generates a small waiting time, equal to 42733 μs for task *A* when it is preempted and 14770 μs for task *B* in the ready queue. This waiting time represents only 16% of the overall running time. For the other tasks, the scheduler response is immediate, and the tasks are performed without preemption or migrations that substantially decrease the configuration overhead estimated in the following stage of resolution. The short response time of the scheduler is also reinforced by parallel computation.

Thanks to parallel computation and the optimization parameters, especially the launching of tasks whenever the start times of their repetitions are valid, the makespan is optimal and equal to 353394 μs. Compared to sequential execution of the DAG, the achieved speedup is 1.03.

This speedup refers to the amount by which the pipelined scheduling speeds up the sequential execution of the DAG. Consequently, the parallel efficiency of this scheduling indicates how efficiently the RZs are exploited to perform DAG execution and is obtained by dividing the achieved speedup by the number of used RZs in the multi-reconfigurable-unit system. In our resolution, the parallel efficiency is about 0.5 which is considered acceptable as tasks are heterogeneous and are not allowed to be executed in all the RZs. The resolution of this first sub-problem is conducted on a CPU of 2 GHz with 2 GB of RAM and lasts 6280 seconds.

4.3. Partitioning/Fitting RZs Results and Placement Quality Evaluation. Based on powerful solvers, RZ_1 and RZ_2 are fitted on their most suitable RPBs defined by the following coordinates after 80.63 seconds on Virtex 5 FX70T (8 × 47 RBs):

(i) RPB_1 for RZ_1: $X_1 = 14$, $Y_1 = 6$, $WRPB_1 = 32$, $HRPB_1 = 6$,

(ii) RPB_2 for RZ_2: $X_2 = 14$, $Y_2 = 3$, $WRPB_2 = 33$, $HRPB_2 = 3$.

Table 4 shows the comparison between the RBs of the obtained RPBs and their RZs expressed in the last column by cost Δ. The not null differences in RBs (Δ) are due to the rectangular shape of the RPBs and the heterogeneity of the device. Δ gives RB_3 in excess. In fact, both RPBs include RB_4, and in Virtex 5 FX70T there is no way to book RBs in a given RPB requiring DSP resources (RB_4) with CLBL and CLBM resources (RB_1 and RB_2) without crossing BRAM (RB_3) columns. Costs Δ also demonstrate how much the resulting placement of required RZs ensures resource efficiency. The low values found in Δ (1 RB_3 and 2 RB_3) show that the obtained RPBs are very close to their corresponding RZs, and consequently, resource efficiency is maintained during the conducted resolution.

Figure 11 depicts the floorplanning of RPB_1 and RPB_2 on Virtex 5 FX70T as obtained by their coordinates. Figure 11 also represents the placing/routing of tasks *A* and *D* named,

Table 4: Resource efficiency.

	RB_1	RB_2	RB_3	RB_4	Δ
RPB_1	8	7	2	1	1 RB_3
RPB_2	9	7	2	1	2 RB_3

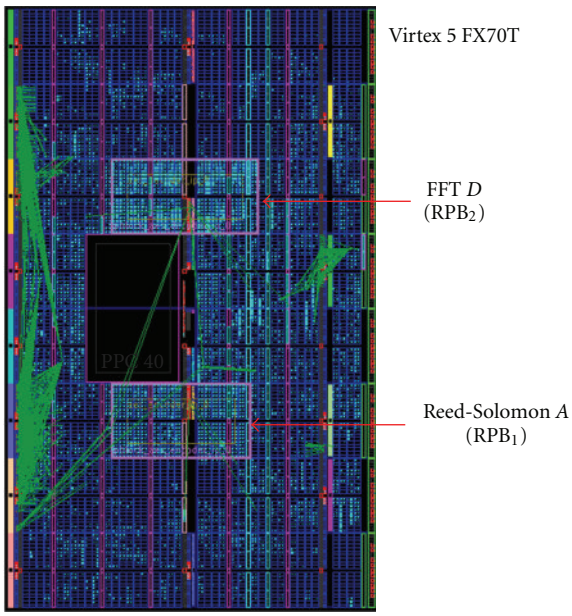


Figure 11: Floorplanning of RPBs and placing/routing of tasks *A* and *D* on Virtex 5 FX70T.

respectively, Reed-Solomon and FFT, which are running, respectively, on RZ_1 and RZ_2 during DAG execution in the time interval [57503, 69309]. The obtained results show an average resource utilization of 12.46% of the available resources on the reconfigurable device. This average is computed according to the number and the cost of each RB type. Optimization in the utilization of resources minimizes the area of the FPGA which is reconfigured at runtime. Due to dynamic partial reconfiguration, resource efficiency is improved by 17.3% compared to the static design of the chosen DAG. The static design is created by floorplanning each task in the DAG on its unique corresponding RPB without sharing any RPBs between different tasks. Once the RPBs are allocated on the reconfigurable device, their bitstreams are created, and their real configuration overheads are computed. The incurred reconfiguration overhead obtained after mapping/scheduling is 6729 μs and represents only 2% of the total running time of the DAG.

Although the experimental conditions are not the same in terms of DAG size and used architecture, we compared our results to attainable improvements in previous works of multiprocessor scheduling. Speedup in our resolution is modest and is about 1.03, compared to [6] which achieves 5.01 for FIR implementation and [3], which reaches a speedup of up to 2. Similarly, regarding parallel efficiency on processors, the results of [6] show a parallel efficiency of 1.3, [3] improves this parameter to 1, and the highest parallelism system is obtained in the work described in [10] which produces 9.3 degrees of parallelism. Nevertheless, our methodology ensures a parallel efficiency of 0.5. The modest improvement in speedup and parallelism results obtained with our methodology can be explained by the heterogeneity of the tasks, the non suitability of all the provided RZs to execute the tasks, and the small number of provided RZs in order to increase resource efficiency in reconfigurable devices. The highest improvement in speedup and parallel efficiency reached in previous works of multiprocessor scheduling does not take into account the resource efficiency and the processor heterogeneity. Effectively, the processors are considered homogeneous as they have identical features, and tasks are allowed to be executed on whatever processor whenever is idle.

However, compared to [18] where 80% of available resources are utilized, our methodology increases resource efficiency in heterogeneous devices by up to 17% compared to a static design and optimizes reconfigurable resource utilization by up to 12.46%. In contrast to [2] that targets an application of two or three tasks and attains a configuration overhead of 8% of total execution time, we immensely reduced the configuration overhead to 2% of the running time for a given DAG of five tasks. Similarly, in [24], Resano et al. attain 18% of configuration overhead to schedule only a JPEG decoder.

Concerning computational complexity, compared to the proposed heuristics to perform multiprocessor scheduling, the worst case temporal complexity is $O(n * (n + e))$, where n is the number of nodes, and e is the number of edges in the graph. We are aware that the efficiency and the optimality of our proposed methodology may be impaired by its temporal complexity in finding the optimal solution and which grows exponentially with the number of tasks in the DAG. It is estimated by: $O(2^{\mathrm{NT}*\mathrm{NZ}*NbrPreemp*\mathrm{HP}*NbrIter}) + O((Device_{width^2} \times Device_{height^2})^{\mathrm{NZ}})$, where *NbrPreemp* and *NbrIter* are, respectively, the maximum number of preemption points and the maximum number of execution iterations for a given task.

Thanks to the pertinent results obtained by the studied DAG, the efficiency of our proposed methodology in performing the placement and scheduling of small DAGs with few tasks is reinforced. However, due to the large size of search space containing the candidate solutions which will be checked for admission by constraints and evaluated for optimality by the objective function, our proposed approach is not capable to deal efficiently with intensive paralleled applications with hundreds of repetitive tasks. Effectively, using our approach in this class of application burdens the resolution time and the memory space, required for the computing of the several constraints and the different objective functions, and for searching the high number of assignments of possible values to the variables, parameters, and indexes. Currently, in our research project FOSFOR (Flexible Operating System FOr Reconfigurable platforms), the proposed methodology is efficiently employed as the target application is of type dataflow and contains small number of tasks.

5. Conclusion and Future Work

Under strict real-time constraints and from a parallel processing perspective, our paper deals with the problem of static scheduling of DAGs on multi-reconfigurable-unit system. In our opinion, most of the works proposed in this field do not consider resource efficiency or configuration overhead, and they are not applied to new heterogeneous technology. The approaches focus only on improving computation speedup and parallel efficiency for DAGs on homogeneous execution units. By means of a new methodology comprising three main stages and by selecting a preemptive model, we take into account periodicity, precedence, dependence, and real-time constraints, and we employ rigorous efficient analytic resolution in order to enhance quality of placement and scheduling in the most recent heterogeneous reconfigurable devices. Our methodology is illustrated in a realistic application, and the results obtained are encouraging. The resolution tries to find the trade-off between all the cited criteria since the performance of the DAG on reconfigurable devices is mainly defined by the degree of parallelism, the resource efficiency, and the amount of incurred reconfiguration. Our proposed approach is largely dependent on the physical features of tasks and technology as well as on temporal characteristics. During our tests, we concluded that our proposed methodology is efficient for small DAGs rather than for larger ones. In fact, solver processing is immensely delayed when the number of tasks in DAG exceeds five.

Static multiprocessor scheduling is a well-understood problem, and many efficient heuristics have been proposed to create compile-time scheduler scenarios. However, the approaches face difficulties in dealing with nondeterministic systems with run-time characteristics that are not well known before the DAG running. Thus, our future challenge is to define dynamic scheduling in heterogeneous reconfigurable devices to be applied for several DAGs of different sizes with nondeterministic behavior. We aim to consider intertask communication, all the specified constraints detailed throughout this paper, as well as to optimize all the cited criteria, especially degree of parallelism, resource efficiency, and configuration overheads.

Acknowledgments

The authors would like to thank the National Research Agency in France and the world-ranking "Secured Communicating Solutions" (SCS) cluster that sponsors our research project FOSFOR (flexible operating system for reconfigurable platforms). This work was also supported by AIMMS technical support and Xilinx tools.

References

[1] G. L. J. Djordjević and M. B. Tošić, "A heuristic for scheduling task graphs with communication delays onto multiprocessors," *Elsevier Parallel Computing*, vol. 22, no. 9, pp. 1197–1214, 1996.

[2] J. A. Clemente, C. Gonzalez, J. Resano, and D. Mozos, "A hardware task-graph scheduler for reconfigurable multitasking systems," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 79–84, Cancun, Mexico, December 2008.

[3] F. E. Sandnes and G. M. Megson, "Improved static multiprocessor scheduling using cyclic task graphs: a genetic approach," in *Parallel Computing: Fundamentals, Applications and New Directions*, vol. 12, pp. 703–710, North-Holland, 1998.

[4] Y. Abdeddaim, A. Kerbaa, and O. Maler, "Task graph scheduling using timed automata," in *Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Processing (IPDPS '03)*, p. 237, Nice, France, April 2003.

[5] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," *International Journal for Reconfigurable Computing*, pp. 97–102, 2008.

[6] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '09)*, pp. 33–38, Nice, France, April 2009.

[7] F. E. Sandnes and O. Sinnen, "A new strategy for multiprocessor scheduling of cyclic task graphs," *International Journal High Performance Computing and Networking*, vol. 3, no. 1, pp. 62–71, 2005.

[8] M. Huang, H. Simmler, P. Saha, and T. El-Ghazawi, "Hardware task scheduling optimizations for reconfigurable computing," in *Proceedings of the 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '08)*, pp. 1–10, Austin, Tex, USA, November 2008.

[9] S. Fekete, E. Kohler, P. Saha, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '01)*, pp. 658–665, Munich, Germany, March 2001.

[10] Z. Pan and B. E. Wells, "Hardware supported task scheduling on dynamically reconfigurable SoC architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, Article ID 4633696, pp. 1465–1474, 2008.

[11] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1235–1238, 1975.

[12] I. Belaid, F. Muller, and M. Benjemaa, "Off-line placement of hardware tasks on FPGA," in *Proceedings of the 19th International Conference on Field Programmable Logic and Application (FPL '09)*, pp. 591–595, Prague, Czech Republic, September 2009.

[13] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[14] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 178, Nice, France, April 2003.

[15] M. Handa and R. Vemuri, "An efficient algorithm for finding empty space for online FPGA placement," in *Proceedings of the Design Automation Conference (DAC '04)*, pp. 960–965, San Diego, Calif, USA, June 2004.

[16] T. Marconi, Y. Lu, K. Bertels, and G. Gaydadjiev, "Intelligent merging online task placement algorithm for partial reconfigurable systems," in *Proceedings of the Design Automation Test Europe Conference (DATE '08)*, pp. 1346–1351, Munich, Germany, March 2008.

[17] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '04)*, p. 134, Santa Fe, New Mexico, April 2004.

[18] H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Task rearrangement on partially reconfigurable FPGAs with restricted buffer," in *Proceedings of the International Conference on Field Programmable Logic and Application*, pp. 379–388, Villach, Austria, August 2000.

[19] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: a survey," *European Journal of Operational Research*, vol. 141, pp. 241–252, 2001.

[20] A. Lodi, S. Martello, and D. Vigo, "Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem," in *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pp. 125–139, Kluwer Academic Publishers, 1997.

[21] M. Panainte, K. Bertels, and S. Vassiliadis, "FPGA-area allocation for partial run-time reconfiguration," in *Proceedings of the Design Automation Test Europe Conference (DATE '05)*, pp. 100–105, Munich, Germany, March 2005.

[22] http://www.aimms.com/.

[23] "Virtex-5 FPGA Configuration User Guide," Xilinx white paper, August 2009.

[24] J. Resano, D. Mozos, D. Verkest, S. Vernalde, and F. Catthoor, "Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Application*, pp. 585–594, Lisbon, Portugal, September 2003.