Resource-Efficient Scheduling for Partially-Reconfigurable FPGA-based Systems

Andrea Purgato, Davide Tantillo, Marco Rabozzi, Donatella Sciuto, Marco D. Santambrogio
Politecnico di Milano, Milan, Italy
{andrea.purgato, davide.tantillo}@mail.polimi.it,
{marco.rabozzi, donatella.sciuto, marco.santambrogio}@polimi.it

***Abstract*—In this paper we present a novel scheduling technique for partially-reconfigurable FPGA-based systems that allows to achieve high quality results in terms of overall application execution time. The proposed algorithm exploits the notion of resource efficient task implementations in order to reduce the overhead incurred by partial dynamic reconfiguration and increase the number of concurrent tasks that can be hosted on the reconfigurable logic as hardware accelerators. We evaluate a fast deterministic version of the scheduler that is able to find good quality solutions in a small amount of time and a randomized version of the approach that can be executed multiple times to improve the final result.**

I. Introduction

In the last few years, System on Chip (SoC) architectures consisting of processor cores tightly coupled with reconfigurable logic have become popular [1]. These SoCs allow the designer to realize complex applications in which hardware software co-design solutions are required to achieve the best performance. Furthermore, current Field-Programmable Gate Arrays (FPGAs) allow to reconfigure portion of the device dynamically while components not affected by the reconfiguration process can still operate [2]. This feature, dubbed as Partial Dynamic Reconfiguration (PDR), allows to virtually increase the resource availability of the FPGA by changing at runtime the set of configured hardware modules. To enable PDR, the reconfigurable logic is partitioned into several *reconfigurable regions* each being able to host different hardware accelerators over time. Each reconfigurable region must be defined large enough to satisfy the resource requirements of the hardware modules that are assigned to it, while the floorplanning of the regions on the FPGA must comply with PDR constraints [3]. The reconfiguration process is then performed by means of a dedicated component, such as the Internal Configuration Access Port (ICAP) available on Xilinx devices, that is exploited to load the partial bitstream for the reconfigurable region to the FPGA configuration memory.

Even though PDR provides a great flexibility for designing the system, the overhead incurred during the reconfiguration process must be carefully taken into account since it can easily jeopardize the performance gain achieved by hardware acceleration [4]. The reconfiguration time is proportional to the amount of resources required by the reconfigurable region in which the new hardware accelerator has to be allocated, it is thus crucial to minimize unused resources across region configurations and mask the reconfiguration time whenever possible.

An additional degree of freedom given to the designer is the selection of the implementation for an application task. Indeed, a task can be either implemented in software and run on the available processor cores, or implemented as a hardware accelerator. Furthermore, multiple hardware implementations with different resource requirements and execution time might also be available. As an example, the designer can provide a software implementation for a given task and leverage High-Level Synthesis (HLS) tools to generate multiple hardware implementations by setting different loop unrolling factors to trade off task performance against resource requirements.

Overall, the resulting task scheduling problem is NP-hard as it represents a more general optimization version of the Resource Constrained Scheduling Problem (RCSP) that is NP-complete [5]. Thus efficient scheduling heuristics are required to both quickly evaluate the potential performance achievable by the application on a given architecture and provide optimized solutions. In this paper, we propose an offline scheduling technique for application taskgraphs on SoCs featuring processor cores and a partial dynamic reconfigurable FPGA. The proposed approach leverages the floorplanning algorithm presented in [3] to ensure that the final solution admits a feasible floorplan. Within this context, we may summarize our contribution as follows:

- we introduce the notion of resource-efficiency as a mean to guide the scheduler in the generation of a suitable set of reconfigurable regions;
- we present a fast deterministic scheduling heuristic for the optimization of the application execution time;
- we propose a randomized version of the scheduling heuristic that can be exploited to trade off algorithm execution time and quality of the final solution;
- finally, the effectiveness of the proposed algorithms are evaluated on a large set of synthetic benchmark and compared with state-of-the-art approaches.

The remainder of the paper is organized as follows: Section II shows the related work in the literature, Section III gives a formal description of the problem, Section IV presents a high level view of the proposed approach, Section V discusses the implementation details of the deterministic scheduler, Section VI shows how we derived the randomized version of the algorithm, Section VII evaluates our approach on different problem

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.176

instances and, finally, Section VIII draws the conclusions.

II. Related work

In the literature, several approaches that address the task scheduling problem on reconfigurable architectures have been proposed ([4], [6]–[13]). However, only few of them explicitly take into account PDR and consider reconfigurations as separate tasks performed by a dedicated component ([4], [6]–[9]).

In [12], the authors propose an exact algorithm for the scheduling and mapping of tasks having a single hardware implementation on FPGAs with homogeneous resources. However, the approach does not consider contention on the reconfigurator, so that an unlimited number of reconfigurations can be performed concurrently. The same limitation still holds for the optimal algorithm described in [13] where reconfigurations are not explicitly handled as separate tasks. Furthermore the approach considers reconfigurable regions having equal dimensions, thus limiting the size of the solution space and leading to potential suboptimal results. Differently from [12] and [13], [10] gives more flexibility in terms of the type of available implementations for the application tasks. In [10], the authors present a hybrid approach based on genetic algorithm and list based scheduling that allows to partition the tasks of the applications either in software or on the FPGA. However, the bottleneck due to the single reconfigurator is still not taken into account. The work developed in [11] generalizes the partitioning of tasks to an arbitrary set of processing elements and adopt clustering techniques to reduce the communication overhead across different components, nevertheless the algorithm does not consider PDR for FPGAs components.

Among the approaches that take into account PDR and contention on the reconfiguration controller, [9] is worth mentioning. The authors in [9] propose the PARLGRAN heuristic, an improved solution with respect to their previous work [7] for the scheduling and placement of tasks considering physical constraints. The approach uses anti fragmentation techniques for the definition of the areas on the FPGA and exploits time slacks within the reconfigurator to mask the reconfiguration overhead. However, the applicability of the algorithm is limited to applications whose tasks have a single hardware implementation. In [8], the authors present an Integer Linear Programming (ILP) formulation that allows to perform scheduling, mapping and define the reconfigurable regions for the tasks assigned to the FPGA. The model considers the reconfigurations as separate tasks and enables schedules in which the configuration of a task does not need to precede immediately its execution. The latter strategy is dubbed as *reconfiguration prefetching* and allows to improve the final solution thanks to the added flexibility for the reconfigurations scheduling. The formulation also consider *module reuse*, that is the possibility to avoid reconfiguration among subsequent tasks that have the same implementation and are scheduled in the same reconfigurable region. Furthermore, [8] generalizes the task scheduling problem by giving the possibility to consider multiple reconfiguration controllers. Nevertheless, the resulting complexity of the ILP formulation makes the approach not viable even for small problem instances.

Both the approaches presented in [4] and [6] explicitly consider partial dynamic reconfiguration as a mean to optimize the execution time of the application. In [4], the authors tackle the scheduling and the mapping of tasks on the architectural components separately. Ant Colony Optimization (ACO) is used to explore the solution space in terms of tasks mapping, while, at each iteration of the metaheuristic a list-based scheduler is invoked to search for an optimized schedule given a fixed mapping. The separation of the two phases allows to manage the complexity of the problem but, at the same time, increases the probability to find suboptimal solutions. On the other hand, in [6] the authors propose a Mixed-Integer Linear Programming (MILP)-based algorithm that tackle simultaneously the mapping of tasks on the architecture and the scheduling of their execution. Even though the approach could potentially be used to search for the optimal solution to the problem, the execution time of the algorithm grows exponentially with the number of tasks, thus making it impractical for large problem instances. To overcome this difficulty the authors propose IS-k, an iterative approach in which k tasks at a time are optimally scheduled exploiting the MILP model. Even though IS-k showed to achieve better results than [4], the iterative scheme of the algorithm makes it vulnerable to initial wrong decisions as demonstrated in Section IV.

Within this work we propose two novel scheduling heuristics that are able to improve the quality of the results achieved by [6] in terms of running time and schedule execution time. The approaches take into account PDR, the reconfiguration overhead caused by contention on the reconfiguration controller and allows the designer to provide a set of different implementations for every single task. Furthermore, we exploit the floorplanning algorithm developed in [3] in order to identify the constraints for the reconfigurable regions and to validate the final solution.

III. Problem description

The task scheduling problem addressed in this work is similar to one presented in [6] and [4]. Given a description of the target architecture in terms of processor cores and the resource availability of the reconfigurable logic, the goal is to schedule the applications tasks either in Hardware (HW) or Software (SW) trying to minimize the overall execution time while possibly exploiting PDR. More formally, the architecture description is given by the following parameters:

$P :=$ set of available processor cores;

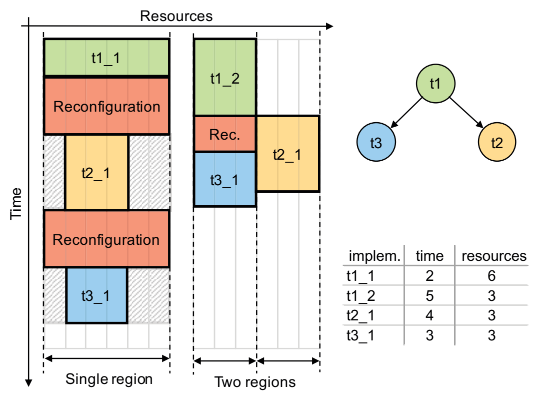
$R :=$ set of reconfigurable resources on the FPGA (e.g.: CLB, BRAM, DSP, ...);

$recFreq :=$ the amount of bitstream that can be reconfigured in a second on the reconfigurable logic;

$maxRes_r :=$ number of resources of type $r \in R$ available on the FPGA;

Notice that within the architecture we consider a single reconfiguration component, thus no two separate reconfigurations can occur at the same time due to contention.

On the other hand, the application is given in terms of a taskgraph, that is a Directed Acyclic Graph (DAG) $G = (T, E)$



implem.	time	resources
t1_1	2	6
t1_2	5	3
t2_1	4	3
t3_1	3	3

Fig. 1. Impact of implementation selection on schedule execution time.

in which a node $t \in T$ represents a specific application task and an arc $(t_1, t_2) \in E$ denotes data dependency between tasks $t1$ and $t2$. Furthermore, each task can have several hardware and software implementations whose details are given by the following parameters:

$I_t^H :=$ set of available HW implementations for task $t \in T$;
$I_t^S :=$ set of available SW implementations for task $t \in T$;
$I_t :=$ set of all implementations for task $t \in T$ $(I_t^H \cup I_t^S)$;
$time_i :=$ execution time of implementation $i \in I_t$;
$res_{i,r} :=$ resources of type $r \in R$ required by hardware implementation $i \in I_t^H$;

The communication overhead among different tasks is not explicitly handled by the adopted problem representation, however the time needed to read and write data for a given implementation can be included within its execution time. Furthermore, we assume that at least a software implementation is available for each task.

The output of the scheduling algorithm consists of: (1) the set of reconfigurable regions together with their resource requirements, (2) a mapping function that assign each task to a specific implementation and to a processor core or a reconfigurable region, (3) the time slot allocated for each task, and, (4) the set of required reconfigurations together with their time slots. Furthermore, the scheduler ensures that the dependencies among different tasks are respected, guarantees that reconfigurations are performed between the execution of tasks assigned to the same reconfigurable region and ensures that the resource required by the reconfigurable regions do not exceed the available FPGA resources.

IV. Proposed approach

A critical step of the task scheduling problem is the selection of the implementation to be used for a specific task. As an example, in Figure 1 we show two possible schedules for a simple application consisting of tasks $T = \{t1, t2, t3\}$ where task $t1$ has two available hardware implementations ($t1\_1$ and $t1\_2$), while tasks $t2$ and $t3$ have a single hardware implementation. Software implementations are not reported, but we assume that their execution time is high enough so that, when selected, they always worsen the overall schedule. The figure also shows the data dependencies between the three tasks and, for the sake of simplicity, considers a single type of FPGA resource. The available implementations for task $t1$ offer a trade off between execution time and resource requirements as it generally happens for real hardware implementations. As we can see form the schedule on the left, the selection of implementation $t1\_1$ provides the best execution time for task $t1$ but leads to the generation of a single large reconfigurable region. This choice is not efficient in terms of resource utilization and worsen the overall execution time in three ways: (1) limits parallelism since a single task at a time can run on a reconfigurable region, (2) increases the number of required reconfigurations and, (3) leads to higher reconfiguration times since a larger bitstream is required for the region reconfiguration. On the other hand, the selection of $t1\_2$ locally worsen the execution time for task $t1$ but eventually improves the schedule execution time, as shown in the schedule on the right.

Iterative algorithms such as IS-k perform subsequent greedy optimizations of the schedule considering k tasks at a time. In the extreme case of IS-1, the implementation $t1\_1$ would be selected, thus increasing the overall application execution time. In order to overcome this issue, we propose a deterministic scheduling heuristic that: (1) starts by considering implementations having a suitable trade off in terms of resource requirements and execution time, (2) assigns the hardware tasks on the reconfigurable logic giving priority to tasks with resource-efficient implementations, (3) creates an initial optimal but generally not feasible schedule and, (4) modifies the initial schedule to achieve a feasible solution by increasing as least as possible the overall execution time. With resource-efficient task implementation we mean those hardware implementations that have a high ratio between execution time and required resources. Such implementations tend to have a high execution time with low area usage and, as shown in Figure 1, they allow to distribute more evenly the load on the reconfigurable logic. In addition, we also propose a randomized variant of the scheduler that relaxes the fixed priority based on resource-efficiency and allows to explore a larger solution space.

Both the proposed algorithms are able to generate directly the set of reconfigurable regions S needed by the hardware implementations and the resource requirements $res_{s,r}$ for each region $s \in S$ and resource type $r \in R$. Such information are also exploited internally by the scheduler to estimate the bitstream size for a reconfigurable region $s \in S$ by means of the following equation:

$$bit_s = \sum_{r \in R} res_{s,r} \cdot bit_r \tag{1}$$

where bit_r represents the average number of bits required to configure a single resource of type r whose value is derived form the number of configuration frames and resources available in a FPGA tile of type r [14]. Starting from the bitstream size, the estimated reconfiguration time for region $s \in S$ is simply computed as:

$$reconf_s = \frac{bit_s}{recFreq} \tag{2}$$

Even though the schedulers guarantee that the resources required by different reconfigurable regions do not exceed the amount of resources available on the FPGA, it is still necessary to verify whether the set of regions admits a floorplan that complies with the PDR constraints [2]. Similarly to [6] a floorplanning algorithm is executed after the scheduling phase to search for a feasible floorplan. If a solution is not found, the scheduler is re-executed by virtually reducing the available FPGA resources $maxRes_r$ by a constant factor.

V. Implementation details

The proposed deterministic scheduling heuristic can be subdivided in a sequence of eight different steps. The first one, dubbed as *implementation selection*, assigns each task to an initial hardware or software implementation based on a cost metric, then, during the subsequent *critical path extraction* step an initial schedule is generated assuming unlimited resources. The third step, named *regions definition*, re-evaluates the initial schedule and assigns each task requiring a hardware implementation to a reconfigurable region by leveraging an efficiency index. After having defined the reconfigurable regions, a post processing step called *software task balancing* tries to improve the schedule by switching tasks implementation from hardware to software. Once the implementations choices have been fixed, step five performs a *start and end time computation* for each task. Then, step *software task mapping* assigns the software tasks to the available processors and step *reconfiguration scheduling* concludes the generation of the schedule by defining the reconfiguration tasks and ensuring that no two reconfigurations overlap in time. Finally, the *feasibility check* step invokes a floorplanning algorithm to verify that the final solution is feasible. The details of the algorithm steps are discussed in the following subsections.

A. Implementation selection

Within this phase, the scheduler identifies and selects the most suitable implementation for each task of the application. As a first step, for every task $t \in T$ a cost is associated to each hardware implementation $i \in I_t^H$:

$$cost_i = \frac{\sum_{r \in R} weightRes_r \cdot res_{i,r}}{\sum_{r' \in R} weightRes_{r'} \cdot maxRes_{r'}} + \frac{time_i}{maxT} \tag{3}$$

where:

$$\begin{aligned} weightRes_r &= 1 - \frac{maxRes_r}{\sum_{r' \in R} maxRes_{r'}} \\ maxT &= \sum_{t \in T} (\min_{i \in I_t} time_i) \end{aligned} \tag{4}$$

As shown in equation 3, the cost of an implementation takes into account the relative amount of resource required on the FPGA, and, the needed execution time normalized with respect to the schedule in which all the tasks are executed in series using the lowest execution time. This allows to assign a higher cost to those implementations that either use a relatively large number of resources or execution time. Notice also that higher importance is given to resources that are more scarce on the device. After having assigned the implementation costs, the algorithm identifies the HW implementation $i_H \in I_t^H$ having the lowest cost and the software implementation $i_S \in I_t^S$ having the lowest execution time. Finally the implementation with the lowest execution time among the SW and HW implementations i_S and i_H is assigned to task t. In the next steps we refer to hardware and software tasks depending on the type of implementation selected.

B. Critical path extraction

Once every task $t \in T$ has been assigned an implementation $i \in I_t$, and consequently, an execution time $T_{EXE_t} = time_i$, the taskgraph topological order is computed and its critical path is extracted using the Critical Path Method (CPM). Every task belonging to the critical path is categorized as critical while all the other tasks are labeled as non critical tasks. The CPM generates a time interval for every task $t \in T$ between a minimum and a maximum time instant defined as a time window $w_t = [T_{MIN_t}, T_{MAX_t}]$, where T_{MIN_t} represents the minimum time at which task t can start its execution, while T_{MAX_t} is the last time instant at which the task execution can be completed without propagating delays to the overall schedule. In the following steps, whenever a task is assigned a different implementation, the time windows are recomputed with respect to the current tasks dependencies.

C. Regions definition

The goal of this phase is to define the set of reconfigurable regions S, specify the resource requirements $res_{s,r}$ for each region $s \in S$ and resource $r \in R$ and assign each hardware task to one of the defined regions. As a preprocessing step, the efficiency index for the each task t for which a hardware implementation i has been selected, is computed as follows:

$$eff_i = \frac{time_i}{\sum_{r \in R}(res_{i,r} * weightRes_r)} \tag{5}$$

Then, the phase starts by setting $S = \emptyset$ and subsequently loops through the hardware tasks trying to assign them to already defined regions, or generating new ones if needed. The order in which hardware tasks are processed determines the set of the reconfigurable regions that are generated and, as a consequence, greatly impacts the quality of the final schedule as discussed in Section IV. In order to reduce the probability to incur in schedule delays, the region assignment is performed for critical tasks first. Among the set of critical and non critical tasks, precedence is given to those having a hardware implementation with higher efficiency index. This choice tends to increase the number of reconfigurable regions that can be defined on the FPGA giving the possibility to better exploit hardware parallelism.

The region assignment for a critical task t having a hardware implementation i is done as follows:

1) We consider the set of regions $S_t \subseteq S$ such that $s \in S_t$ has the following properties: s has enough resources to host t, the time windows of tasks already assigned to s do not overlap with w_t and with the time window of the reconfiguration required to host t. If $S_t \neq \emptyset$, then task t is assigned the region $s \in S_t$ having the lowest bitstream bit_s.

2) else, if there are enough available resources on the FPGA, a new region s with resource requirements $res_{s,r} = res_{i,r}$ is added to S and assigned to task t.
3) Otherwise, the implementation of task t is switched to the fastest SW one and the time windows are updated.

Area assignment for a non critical task t with hardware implementation i follows a similar procedure, however the goal is shifted towards trying to maximize the FPGA utilization:

1) If there are enough available resources on the FPGA to host implementation i, a new region $s \in S'_t$ with resource requirements $res_{s,r} = res_{i,r}$ is added to S and assigned to task t.
2) else, we consider the set of regions $S'_t \subseteq S$ such that s has enough resources to host t and the time windows of tasks already assigned to s do not overlap with w_t. If $S'_t \neq \emptyset$, then task t is assigned to the region $s \in S'_t$ having the lowest bitstream bit_s.
3) Otherwise, the implementation of task t is switched to the fastest SW one and the time windows are updated.

Notice that in the previous step, whenever a task is assigned to an existing reconfigurable region, new dependencies are inserted into the taskgraphs to guarantee the ordering of tasks inside each reconfigurable region.

D. Software task balancing

Within the previous phase some of the task implementations could have been switched to SW and, as a consequence, the overall solution could have worsened since SW implementations tend to have a higher execution time than the HW ones. This change in the implementation choice generally produces a solution in which HW tasks are blocked for a long time waiting for the completion of other SW tasks. In order to exploit the underutilized resources on the FPGA, it is possible to check if some SW tasks can be executed in hardware during the unused time interval. The SW task balancing procedure is applied to all the SW tasks $t \in T \mid I_t^H \neq \emptyset$ starting from the ones having lower T_{MIN_t} as follows:

1) An estimation of the total amount of time spent for reconfigurations is calculated using the equation:
$$totRecTime = \sum_{s \in S} reconf_s \cdot (|T_s| - 1) \qquad (6)$$
where T_s represents the set of hardware tasks assigned to the reconfigurable region s.
2) If $T_{MIN_t} > totRecTime$ and exists a region $s \in S$ whose tasks time windows do not overlap with w_t, then task t is assigned to region s using the hardware implementation with the lowest cost and the time windows are recomputed.

In the above procedure, the total reconfiguration time is used to verify that the task implementation can be moved to hardware without generating contention on the reconfigurator.

E. Start and end time computation

After having fixed all the implementations and assigned the hardware tasks to reconfigurable regions on the FPGA, this phase computes the start and end time for each task $t \in T$ as follows:
$$\begin{aligned} T_{START_t} &= T_{MIN_t} \\ T_{END_t} &= T_{START_t} + T_{EXE_t} \end{aligned} \qquad (7)$$
Notice that the execution time T_{EXE_t} is known from the implementation selected in the previous phases.

F. Software task mapping

The aim of this step is to bind each SW task to one of the available processors in P. The main idea is to consider the software tasks in chronological order and assign them to the processor in which the minimum delay is generated. For each task $t \in T$ the following procedure is applied starting from tasks having the lower T_{MIN_t}:

1) For every processor $p \in P$ the potential delay λ_p is computed as:
$$\lambda_p = min\{0,\ max_{t2 \in T_p}(T_{END_{t2}} - T_{MIN_t})\}, \qquad (8)$$
where T_p is the set of current tasks assigned to p.
2) The task is assigned to the processor $p \in P$ having the lowest delay λ_p and new dependencies are added to ensure task ordering within the processor.
3) The task start and end time are recomputed as:
$$\begin{aligned} T_{START_t} &= T_{MIN_t} + \lambda_p \\ T_{END_t} &= T_{START_t} + T_{EXE_t} \end{aligned} \qquad (9)$$
4) if $T_{END_t} > T_{MAX_t}$ the delay $T_{END_t} - T_{MAX_t}$ is propagated over the taskgraph.

G. Reconfigurations scheduling

The last step required to obtain a complete application schedule is to generate the set of reconfiguration tasks RT for the regions hosting multiple implementations. A reconfiguration occurs between each couple of subsequent tasks $t_{in}, t_{out} \in T$ assigned to the same reconfigurable region and it is needed to load the partial bitstream for task t_{out}. Tasks t_{in} and t_{out} are dubbed as ingoing and outgoing task respectively, if a reconfiguration has an outgoing critical task, then the reconfiguration task is also considered critical. Similarly to the application tasks, each reconfiguration $t \in RT$ has a time window in which it has to be executed to avoid the generation of delay:
$$\begin{aligned} T_{MIN_t} &= T_{END_{t_{in}}} \\ T_{MAX_t} &= T_{START_{t_{out}}} \end{aligned} \qquad (10)$$
On the other hand, the execution time of the reconfiguration t depends on the region $s \in S$ in which the ingoing and outgoing tasks are scheduled:
$$T_{EXE_t} = reconf_s \qquad (11)$$

Critical reconfigurations are scheduled before non critical ones since their delays would be completely propagated on the schedule. For each critical reconfiguration $t \in RT$, the following procedure is applied starting from the reconfiguration having lower T_{MIN_t}:

1) The time window for the reconfiguration task is recomputed since some delays may have modified the time windows of its ingoing and and outgoing tasks.
2) The last scheduled reconfiguration task tl on the reconfiguration component is extracted.
3) If $T_{MIN_t} > T_{END_{tl}}$, then the start time of the reconfiguration is set as $T_{START_t} = T_{MIN_t}$ otherwise the reconfiguration start time is computed as $T_{START_t} = T_{END_{tl}} + 1$.
4) The end time for the reconfiguration is set as usual as $T_{END_t} = T_{START_t} + T_{EXE_t}$
5) Check if the reconfiguration task generates a delay (i.e. $T_{END_t} > T_{MAX_t}$) and, if so, propagate the delay over the taskgrpah.

The scheduling of non critical reconfiguration task is more complex since an existing partial scheduling is already present and we must ensure that no two reconfigurations overlap in time. For each non critical reconfiguration $t \in RT$, the following procedure is applied starting from the reconfigurations having lower T_{MIN_t}:

1) If the time instant T_{MIN_t} lies within the execution of an already scheduled reconfiguration, T_{MIN_t} is set to the first time instead ahead in time in which no other reconfigurations are performed.
2) The start time and end time of the reconfiguration are computed as $T_{START_t} = T_{MIN_t}$ and $T_{END_t} = T_{START_t} + T_{EXE_t}$.
3) If $T_{END_t} > T_{MAX_t}$, the delay of the reconfiguration is propagated to the outgoing task.
4) Finally, if the reconfiguration overlap with the execution of other reconfigurations, those are shifted ahead in time and their delay is propagated over the taskgraph.

H. Feasibility check

At this stage, a schedule and mapping of all the tasks on the target architecture has been performed, however, there is no guarantee that the identified reconfigurable regions admit a valid floorplan on the FPGA. For this purpose, the set of regions S together with the resource requirements $res_{s,r}$ are given has input to the MILP-based floorplanning algorithm presented in [3]. It is worth noting that no objective function is specified for the floorplanner since we are interested in verifying the existence of a solution in a small amount of time. If it is not possible to find a feasible floorplan, the scheduling algorithm is restarted by virtually reducing the number of resources available on the FPGA by a constant factor.

VI. Randomized scheduler variant

The order in which non critical tasks are considered during the reconfigurable region definition presented in Section V-C leads in general to suboptimal schedules, since it does not take into account the impact of tasks dependencies, reconfiguration constraints and the possibility that some task implementations can be switched to SW. However, finding the best ordering is a computationally expensive processes, while sorting the tasks with respect to the efficiency index of theirs selected implementations provides good quality results in a small amount of time. In this section we relax the fixed processing ordering for non critical hardware tasks and we propose a variant of the algorithm that exploits a randomized order. This gives the possibility to explore a larger solution space by executing the scheduler several times, so that the designer can trade off the quality of the solution against the execution time of the algorithm. Furthermore, this randomized approach allows to amortize the computational cost of the floorplanner over different scheduling iterations. A high level view of the scheduler variant is given by Algorithm 1.

```
Function RScheduler(instance, timeToRun) : schedule
    deadline = $getCurrentTime()$ + timeToRun;
    bestSchedule = null;
    bestExeTime = $+\infty$;
    tasksOrder = "RANDOM";
    while deadline $\geq getCurrentTime()$ do
        schedule = $doSchedule(instance, tasksOrder)$;
        if schedule.exeTime $<$ bestExeTime then
            feasible = $checkFloorplan(solution)$;
            if feasible == true then
                bestSchedule = schedule;
                bestExeTime = schedule.exeTime;
            end
        end
    end
    return bestSchedule;
end
```

Algorithm 1: Randomized task scheduling algorithm

The $doSchedule$ function represents the core of the algorithm described in section V devoid from the feasibility check phase, where an additional parameter has been added to specify the type of ordering for non critical hardware tasks during the definition of the reconfigurable regions. On the other hand, the $checkFloorplan$ method is used to query the floorplanner and verify that the current schedule is valid. Overall, the algorithm takes as input the problem instance describing the target architecture together with the application taskgraph and a time budget, while it produces as output the best schedule found. As shown in Algorithm 1 the floorplanner is executed only when a potential improving schedule is found so that the overall time spent in validating the solution on the FPGA is reduced. Furthermore, differently from the deterministic approach, unfeasible solutions in terms of the resulting floorplan are simply discarded without the need to restart the scheduler reducing the amount of available FPGA resources.

VII. Experimental evaluation

Within this section we evaluate the proposed scheduling algorithm with task ordering based on efficiency index (PA), and, the scheduler variant that exploits randomized tasks ordering (PA-R). The achieved results are also compared to the ones found by IS-k [6], since it is the approach that is more close to our context, being able to exploits PDR and giving the possibility to trade off the execution time of the

TABLE I
ALGORITHMS EXECUTION TIME

# Tasks	PA [s]			IS-1 [s]	PA-R / IS-5 [s]
	scheduling	floorplanning	total		
10	0.070	0.332	0.402	1.211	4.734
20	0.097	0.526	0.623	3.286	68.387
30	0.118	0.979	1.097	13.628	90.304
40	0.139	1.074	1.213	25.786	149.460
50	0.161	1.028	1.189	57.215	135.362
60	0.180	1.005	1.185	120.131	189.140
70	0.197	1.091	1.288	256.967	413.137
80	0.216	1.166	1.382	276.271	288.639
90	0.236	0.981	1.217	328.214	288.528
100	0.276	1.041	1.317	564.855	563.129

algorithm against the quality of the solutions. Specifically, the comparisons are performed against IS-1 and IS-5, where IS-1 represents the version of IS-k that has the lowest execution time for small to medium taskgraphs, while IS-5 is able to give better results at the cost of a generally higher but still acceptable execution time.

A. Testing Environment

The evaluations were performed on a test suite consisting of 100 pseudo-random taskgraphs that are organized in 10 groups containing 10 taskgraphs each. The taskgraphs within a group have a fixed number of tasks that varies in the range $[10, 100]$ across different groups. Each task has one software implementation and 3 hardware implementations with heterogeneous resource requirements (i.e. different requirements of CLBs, DSPs and BRAMs). Moreover, we considered that different tasks can share a common implementation so that module reuse can be exploited by IS-k, a feature currently not supported by the proposed approach. The target architectures for the applications is the ZedBoard with Zynq$^{\text{TM}}$-7000 All programmable SoC that provides a dual core ARM Cortex-A9 CPU and a Xilinx XC7Z020 FPGA.

All the tests were run on a Intel Core i7 4700MQ under Linux, while we used Gurobi 6.5 [15] for solving the MILP models for the floorplanner [3] and IS-k [6]. PA, IS-1 and IS-5 were executed until completion, while PA-R was assigned a time budget equal to the time used by IS-5 in order to have a fair comparison in terms of computational efficiency between the two approaches.

B. Results analysis

Figure 2 shows the average schedule execution time for the solutions found by the proposed algorithms and by IS-k with respect to each group of taskgraphs, whereas Table I reports the execution time of the different algorithms, in which the elaboration time of PA has also been subdivided in time spent during the the scheduling and the floorplanning phase. In the following discussion we focus our attention on the comparison between PA and IS-k in terms of both algorithms running time and quality of the achieved results and between PA-R and IS-5 in terms of computational efficiency.

As we can see from Table I, the time required by PA to compute the schedule grows almost linearly with respect

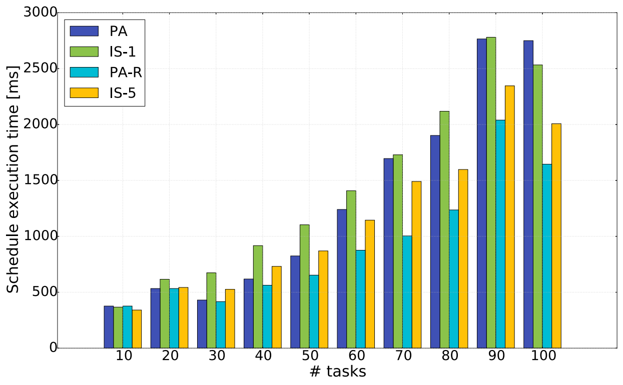


Fig. 2. Comparison between solutions

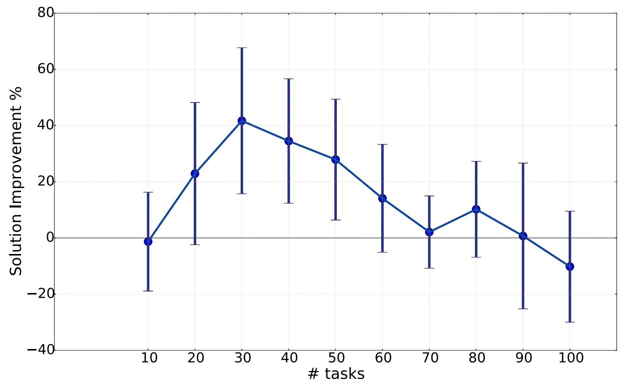


Fig. 3. Average solutions improvement of PA with respect to IS-1.

to the number of application tasks, while the overall PA execution time, that includes also floorplanning, is two order of magnitudes smaller than the one of IS-1 and IS-5 for taskgraphs consisting of 60 or more tasks. In order to better analyze the difference in terms of quality of the solutions, we report in Figure 3 and Figure 4 the average improvements of the schedules execution times achieved by PA using IS-1 and IS-5 as baselines respectively.

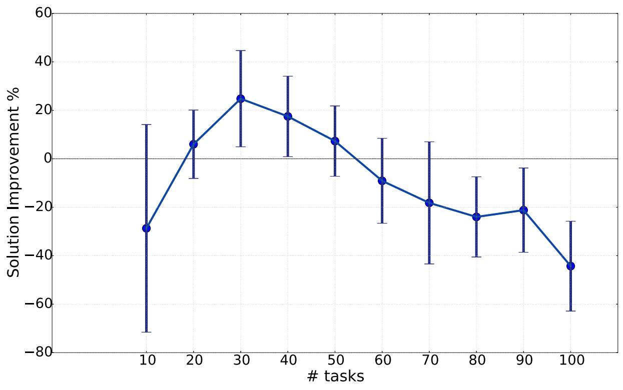


Fig. 4. Average solutions improvement of PA with respect to IS-5.

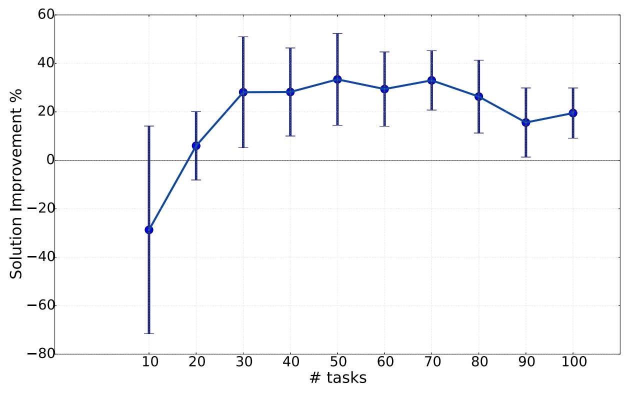


Fig. 5. Average solutions improvement of PA-R with respect to IS-5.

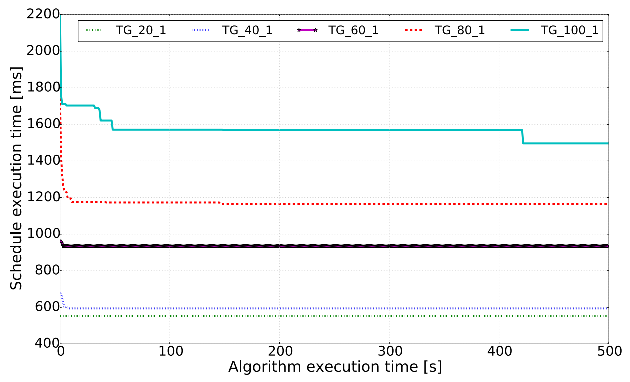


Fig. 6. Solution improvement over time for PA-R on different taskgraphs.

As shown in Figure 3, PA is able to improve IS-1 solutions by 14.8% on average, while the best results are achieved for medium-sized applications having a number of tasks in the range $[20, 60]$. The improvement is mainly obtained by considering hardware implementations having a low impact on the FPGA resources and by scheduling the tasks on the reconfigurable regions giving priority to those tasks whose implementations have higher efficiency indexes. However, for applications with a small number of tasks, there is less contention on the FPGA and thus the benefits of the proposed scheduler are less evident. Notice also that IS-1 tends to reduce the improvement gap of PA for large taskgraphs, this is due to the fact that IS-k spends an exponentially larger amount of time in optimizing the schedule when the number of tasks increases. Indeed IS-k is not a pure greedy algorithm and some of the time variables involved in its MILP formulation are allowed to be modified between subsequent scheduling iterations, thus leaving room for exploring a larger solution space. Overall, PA has a much lower running time than IS-1 and is in general able to improve IS-1 solutions. On the other hand, as shown in Figure 4, the improvement of PA with respect to IS-5 is reduced due to the additional flexibility given to the IS-5 MILP model that also translates in increased running time.

When the goal is shifted towards finding near optimal solutions, the randomized approach PA-R can be applied to trade off the small running time of PA to search for better schedules. Figure 5 shows the improvement of PA-R over IS-5 solutions where both algorithms have been executed for the same amount of time. For small applications consisting of 10 taskgrpahs, IS-5 is still able to provide better solutions by exploiting implementations having low execution time and high impact on the FPGA resources. However, for applications having more than 20 tasks FPGA contention becomes more relevant and PA-R is able to provide an average 22.3% improvement with respect to IS-5 by considering cost effective implementations.

It is worth noting that the high standard deviation shown in Figures 3, 4 and 5 is due to the fact that the number of tasks is not the only factor that influence the improvement achieved by the proposed approach. Further investigations are required in order to clearly identify these factors, however, we observed that the improvements achieved by PA-R with respect to IS-5 are more restrained when either the taskgraph expose a reduced level of parallelism or, at the opposite, when a great proportion of the application tasks can be executed in parallel.

As a final analysis, we executed PA-R with an extended time limit of 1200 seconds over 5 of the given pseudo-random taskgraphs having a number of tasks in the range $\{20, 40, 60, 80, 100\}$. Figure 6 reports the best schedule execution time found by PA-R with respect to the running time of the algorithm. Notice that only the first 500 seconds of execution are shown since we observed that after this time interval the solutions found by PA-R did not improve further. As we can see from the figure, the algorithm generally converges to a good solution in a small amount of time and the convergence time increases together with the number of tasks involved in the taskgraph.

VIII. Conclusions

Within this paper we proposed a novel scheduling approach for applications described in terms of DAG of tasks for SoC featuring homogeneous processing cores coupled with a partial dynamic reconfigurable FPGA. Two different algorithms were presented: the first one, is a deterministic scheduling heuristic that allows the designer to obtain a fast evaluation of the design performance on the target architecture, while the second one exploits randomization to explore a larger solutions space and is able to find optimized results. On average, For medium to large size applications, the proposed randomized algorithm reduces the execution time of the schedules by 22.3% with respect to the MILP-based approach presented in [6] that already proved to achieve better performance than [4].

Future work will investigate the possibility to leverage module reuse in order to further improve the solutions by removing the reconfiguration overhead for tasks sharing the same hardware implementations. Furthermore, we will also consider the possibility to explicitly model the communication overhead between the application tasks to improve the schedule accuracy.

References

[1] Xilinx Inc., "Zynq-7000 all programmable SoC: Technical Reference Manual," 2015. [Online]. Available: http://www.xilinx.com

[2] Xilinx Inc, "Vivado Design Suite User Guide: Partial Reconfiguration," 2014.

[3] M. Rabozzi, A. Miele, and M. D. Santambrogio, "Floorplanning for partially-reconfigurable FPGAs via feasible placements detection," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23nd Annual International Symposium*, 2015, pp. 252–255.

[4] R. Cattaneo, R. Bellini, G. Durelli, C. Pilato, M. Santambrogio, and D. Sciuto, "Para-sched: A reconfiguration-aware scheduler for reconfigurable architectures," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 243–250.

[5] M. R. Garey and D. S. Johnson, *Computers and intractability*. freeman San Francisco, 1979, vol. 174.

[6] E. Deiana, M. Rabozzi, R. Cattaneo, and M. Santambrogio, "A multi-objective reconfiguration-aware scheduler for fpga-based heterogeneous architectures," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*. IEEE, 2015.

[7] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 11, pp. 1189–1202, Nov 2006.

[8] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," *Int. J. Reconfig. Comput.*, vol. 2009, pp. 7:1–7:12, Jan. 2009.

[9] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures," in *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27, 2006*, 2006, pp. 491–496.

[10] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of ProRISC*. Citeseer, 2000, pp. 405–411.

[11] Y. M. Lam, J. Coutinho, W. Luk, and P. H. W. Leong, "Mapping and scheduling with task clustering for heterogeneous computing systems," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 275–280.

[12] S. Fekete, E. Kohler, and J. Teich, "Optimal fpga module placement with temporal precedence constraints," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, 2001, pp. 658–665.

[13] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh, "An optimal algorithm for minimizing run-time reconfiguration delay," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, pp. 237–256, 2004.

[14] K. Vipin and S. A. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *Proc. Intl. Conf. on Reconfigurable Computing: architectures, tools and applications (ARC)*, 2012, pp. 13–25.

[15] I. Gurobi Optimization, "Gurobi optimizer reference manual," 2015. [Online]. Available: http://www.gurobi.com