2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

PaRA-Sched: a Reconfiguration-Aware Scheduler for Reconfigurable Architectures

Riccardo Cattaneo[1], Riccardo Bellini[1], Gianluca Durelli[1],
Christian Pilato[2], Marco D. Santambrogio[1], Donatella Sciuto[1]

[1]Politecnico di Milano, Dipartimento di Elettronica ed Informazione, Milano, Italy,
riccardo.bellini@mail.polimi.it, {rcattaneo,durelli,santambr,sciuto}@elet.polimi.it

[2]Columbia University, Department of Computer Science, New York, NY, USA
pilato@cs.columbia.edu

***Abstract*—Harnessing the full capabilities offered by reconfigurable hardware is still a demanding task: the lack of proper methodologies and the intrinsic time consuming and error prone tailoring of these systems around the specific application places a barrier to the adoption of this technology. Partial and Dynamic Reconfiguration (PDR), in this context, is a specific feature whose potential is undiscussed but yet to uncover.**

In this work, we propose PaRA-Sched, an improvement for a state of the art, highly automated design methodology that allows the designer to rapidly explore the impact of PDR employment during the early stages of the design process. Specifically, we extend the scheduling infrastructure of the framework to explicitly take into account PDR to better explore the design space and improve overall performance by automatically masking reconfiguration time when possible. We show how this additional degree of freedom leads to designs whose performance are improved with respect to the baseline, with a limited increase in time spent during DSE.

***Index Terms*—Field Programmable Gate Arrays, High Performance Computing, Reconfigurable Architectures, Scheduling algorithms, Design methodology**

I. Introduction

MPSoCs are the mainstream platform in today's digital embedded systems market. They combine several general and specific purpose processors into a single chip to deliver improved performance for a specific market need. As the design of MPSoC platforms is a complex, involved, and error prone task, and since the rationale behind the design of a new MPSoC is frequently tied to a company's goal of accelerating an existing software application, many industrial–level tools have been developed in the previous years to effectively support engineers in the design process of these Systems-on-Chip. Some of these (namely: Xilinx Vivado HLS, Synopsys C Compiler, Cadence C-to-Silicon, Calypto CatapultC) have become very popular among the hardware/software engineering community as they allow designers to rapidly derive RTL descriptions of complex software descriptions, in the form of an IP core tailored of the specific vendor platform. Advanced simulation and validation platforms allow designers to accurately estimate during the early design phases both various expected metrics of such platforms.

While ASIC MPSoCs are nowadays mainstream platform, also others benefit from such design tools. For instance, Field Programmable Gate Arrays (FPGAs) can be much easily programmed using these platforms. As one of the major limiting factors to the employment of FPGAs as mainstream computation platforms is the difficulty at programming them using one of the many Hardware Description Languages, these tools play a fundamental role in their larger adoption as mainstream heterogeneous processors.

One specific feature of FPGAs is the ability to reconfigure at runtime part of the implemented logic: this feature is known as *Partial Dynamic Reconfiguration* (PDR) [1]. It offers the possibility to reuse part of the logic across different tasks, effectively sharing resources by time multiplexing them. This is one of the features that make FPGAs very attractive solutions to accelerate portions of a complex target application in hardware at low costs. However, this technique induces several design challenges that must be properly considered while designing such systems.

One major concern when designing reconfigurable systems is that the target architecture where tasks are run is known in advance, i.e. is fixed at design time. This is the result of an estimate of the requirements of the application, but is also a hard constraint on the structure of the platform itself, one that might lead to sub optimal solutions. When reconfiguration – and in particular, when PDR – is employed, this problem is exacerbated [2]. The choice of schedule of these tasks on a given architecture is crucial, too.

Another concern relates to the time it takes for the FPGA to reconfigure the portion of logic targeted by PDR. The bitstream associated to that area must be loaded from a memory into the reconfiguration controller (be it internal or external or tightly coupled with an on-chip hard processor) and then sent to the configuration banks. Depending on the application and frequency of reconfiguration, this might or might not be negligible. As there are few methodologies explicitly considering PDR in the design flow, it is not easy to do an early estimate of the benefits due to PDR or the associated penalties in case it is applied incorrectly [3]. As PDR introduces latencies, specifically, potentially unnecessary ones, a good task scheduler must be able to properly reorder the execution of tasks so as to maximise the amount of "masked" reconfiguration time (i.e.: reconfiguration time not

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.32

spent on the critical path of the application).

Another major concern is due to the impact of a proper interconnection subsystem and topology. As data transfers between tasks (which translates to data communication among processors) is a crucial aspect of the final design [4], the sequence of communications and the topology of the interconnection network (e.g., bus, NoC, FIFO) must be properly taken into account. This is better accomplished with a scheduler that explicitly takes into account the specific needs of the communication tasks.

While there exist some methodologies that ease the development phase of FPGA-based reconfigurable systems, none seems to properly consider the impact of PDR during design phase, lest for A2B [5]. This methodology does explicitly take into account many aspects related to the design of reconfigurable systems from the definition of an architectural template to the simultaneous mapping and scheduling of tasks onto it.

While the authors demonstrated the effectiveness of it, the scheduler that was employed can be improved in at least three ways, that make up the contributions of this work:

- explicit consideration of communication and reconfiguration tasks and masking of the overhead introduced;
- fast qualitative evaluation of a given mapping;
- capability to embed general knowledge about the problem by means of heuristics.

In this paper, we propose *PaRA-Sched* (Partial Reconfiguration-Aware Scheduler), a novel scheduling algorithm that aims at addressing the above cited limitations. It combines different heuristics, inspired by the heuristic algorithm proposed in [6], while considering the possibility to apply *module reuse*; hence, modules that have been already placed on the FPGA are reemployed instead of executing unnecessary reconfigurations. Moreover, PaRA-Sched tries to embed general knowledge about the problem structure so as to make a good choice at a given moment, instead of performing time-consuming complex explorative or evolvable heuristics, which might lead to better solution but heavily affect the performances of the design space exploration algorithm.

The rest of the paper is structured as follows. Section II overviews state of the art scheduling algorithms and their relationship with this work, while highlighting the contributions. Section III briefly describes the framework we extend with the proposed solution, whose organisation and implementation is discussed in Section IV. Section V presents the experimental evaluation of the proposed approach, while Section VI concludes the paper outlining the future improvements of the work.

II. Related Work

Many techniques have been developed in the past to solve the problem of scheduling tasks on a set of resources, also called RCSP (Resource Constrained Scheduling Problem), that is a well known NP-hard optimization problem [7]. Therefore, finding an exact solution for an RCSP is impractical and too complex for real workloads. The different algorithms proposed and outlined in this overview fall in the following categories:

- Exact algorithms, such as Integer Linear Programming (ILP) formulations or frameworks that by means of mathematical structures lead to optimal solutions;
- Heuristic algorithms, that in general lead to suboptimal solutions rather than optimal ones. These algorithms are additionally divided according to their internal functioning:
  - List-based heuristics, which schedule task using a list based on some priority (for example critical path priority);
  - Metaheuristics, such as tabu search, simulated annealing or other methods which can be evolutionary (for example genetic algorithms) and/or naturally inspired (for example ant colony optimization and simulated annealing)

A. Exact algorithms

An exact algorithm to solve the task scheduling problem on dynamically reconfigurable devices has been proposed in [8]. In this paper, Fekete et al. [8] are the first to model communications between different modules, including the communication delay into the execution time of a task. However, this model has the limitation of assuming that an infinite number of reconfigurations can be executed at the same time; moreover, configuration prefetching is not taken into account, since reconfiguration time is just added to the execution time of a task. Therefore, the proposed strategy is not applicable when resource contention of the reconfiguration controller and non negligible reconfiguration overheads occur. Banerjee et al. [6] propose an ILP formulation that tries to overcome the previously mentioned limitations by taking into account configuration prefetching and HW-SW communications introduced, and solves mapping and scheduling at the same time, but this model is suitable only for 1D partial dynamic reconfigurable architectures with a single reconfiguration controller, a very simple architecture compared to the devices currently available. Redaelli et al. [9] propose an ILP formulation tailored to reconfigurable architectures with bi-dimensional partial dynamic reconfiguration, taking into account configuration prefetching, module reuse and anti-fragmentation techniques, as well as the possibility to consider more than one reconfiguration controller. However, limitations due to the huge execution time of an ILP solver with such a complex model still hold, reaching up to 39 days of execution time to solve an instance of ten tasks on an FPGA with 5 rows, 5 columns and 2 reconfigurators. In the following section a brief overview on the heuristic algorithms is provided, to better highlight their benefits with respect to the aforementioned limitations of exact algorithms.

B. Heuristic algorithms

Among the variety of heuristic algorithms that have been developed to solve the task scheduling problem, some of these consider also HW-SW partitioning as a goal: Mei et al. [10] propose an approach based on the combination of a standard genetic algorithm with an improved list scheduling algorithm. Their approach targets dynamically reconfigurable

devices and takes into account HW-SW partitioning and partial reconfiguration overhead; nevertheless, they do not consider configuration prefetching so as to hide as much as possible the reconfiguration time, and the scheduling phase, which means that unfeasible solutions may be generated. Banerjee et al. [6] overcome this limitation by considering simultaneous HW-SW partitioning and scheduling of tasks by means of a KLFM-based heuristic [11], [12]. Once a task is locked for scheduling, it is immediately placed on the device, to ensure correctness by construction of all the schedules. However, as in the case of the ILP formulation proposed in the same work, the target architecture features just one-dimensional reconfiguration controller. Moreover, this algorithm does not consider the possibility of exploiting module reuse, which we do and thanks to which better results are generally obtained. The same authors introduced PARLGRAN [13] in a following work, as an improved solution to scheduling and placement problem with physical constraints. PARLGRAN selects the best granularity of data parallelism for data parallel tasks and takes into account the reconfiguration overhead and possible placement issues, by using two techniques: simple fragmentation reduction consists in placing a new task on the FPGA in the opposite side with respect to the previously scheduled task; exploiting slack in reconfiguration controller postpones the reconfiguration of a task if this reconfiguration introduces a delay in the execution time of the subsequent task, which means that the reconfiguration is non-maskable. This means that we can delay or anticipate a reconfiguration whenever possible - while maintaining order correctness between tasks - in order to reduce total execution time. The drawback of this work is the lack of taking into account memory management of parallel tasks that operate on different chunks belonging to the same data. Another heuristic approach was proposed by Redaelli et al. [9]; they introduced Napoleon, a heuristic reconfiguration-aware scheduler targeted for architectures with 2D partial dynamic reconfiguration. Napoleon uses different techniques to reduce the makespan of the generated schedule, such as configuration prefetching, module reuse and anti-fragmentation during the placement of tasks; in particular, it adopts limited reconfiguration and farthest placement criterion, to increase the probability of module reuse and facilitate the placement of large modules in the center of the FPGA. This approach, however, does not consider explicitly communications between tasks and the possibility of having tasks executed on a processor with a software implementation, besides tasks accelerated on hardware.

As outlined in the previous overview of the state of the art, both exact and heuristic algorithms proposed have limitations or do not apply correctly to the workflow of the toolchain. ILP formulations are characterised by huge complexity, especially if dealing with simultaneous mapping and scheduling problem, hence are too complex to solve real instances in a reasonable time. Heuristic approaches are in general better from an execution time perspective. However, they are characterised by the same limitations as exact algorithms, in fact they fail at considering some key features and aspects, such as configuration prefetching, communications from tasks to memory and from memory to tasks, delays introduced by reconfigurations,

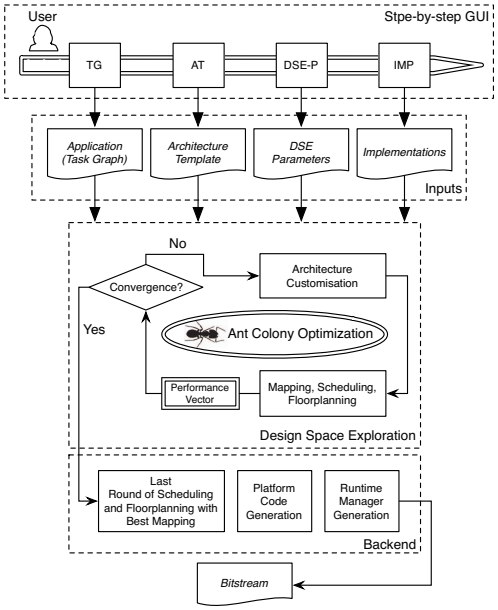


Fig. 1. A2B methodology overview. Observe that the iteration phase realises a DSE exploration step where mapping and scheduling are considered altogether.

bottleneck on the reconfiguration controller (i.e. only one region can be reconfigured at once) and the possibility of reusing already configured modules. Moreover, in most of the proposed approaches, architectures with bi-dimensional partial dynamic reconfiguration are rarely considered, and mapping and scheduling problems are solved simultaneously by the algorithm, increasing the complexity of the problem. Instead, it should be desirable for the toolchain to provide an evolvable mapping algorithm and a fast baseline scheduler which provides an estimation of the makespan based on the current mapping. In the following section a possible solution to this problem will be proposed.

III. A2B Framework Overview

Among the various state-of-the-art methodologies for the design of FPGA-based reconfigurable and heterogeneous systems, the only one we are aware of to explicitly support PDR opportunities while doing mapping and simultaneous scheduling during the design phase is A2B [5]. For this reason, we decided to build upon the existing code base and integrate our scheduling algorithm in this infrastructure. In the following section we present the peculiarities of this framework.

This design methodology allows the user to describe an application in terms of its task graph, an architectural template (i.e. a customisable skeleton of the final architecture) and a set of parameters configuring the toolchain itself, and to rapidly explore different designs providing her with a set of interconnected tools that realise a Design Space Exploration (DSE) cycle aimed at porting a whole software application on a reconfigurable, heterogeneous, FPGA-based platform. Depending on the specific partitioning strategy, exploration parameters and architectural skeletons, different (high quality, as we will see in Section V) solutions are rapidly computed.

This methodology has a working implementation in the form of a graphical user interface (called A2B) that allows the designer to easily interact with the underlying tools in a point-and-click fashion, which improves the overall productivity and learning curve of the framework.

To use this design tool, the user has to input the application in the form of a task graph, which is defined as a Directed Acyclic Graph (DAG) whose vertices represent tasks (abstract stages of the overall computation/application) and edges represent data transfers among tasks.

Once the application is input to the framework, the designer has to specify the architectural template. An architectural template is a "guide" for the framework suggesting the structure of the final solution in terms of interconnections, number of soft and hard processors, and static and reconfigurable processors. Moreover, a template also suggests an initial solution (i.e.: an initial number of customisable components) and their interconnections, in terms of buses and memory elements. This template will then be customised as part of the design space exploration phase.

After specifying the architectural template, the designer must specify the so-called implementations: since the flow allows for more than one kind of processor to be present in the system at the same time, it is mandatory to be able to specify different "ways" to realise (i.e.: compute) the same task (i.e.: a portion of the application) on any meaningful element of the architectural template. For example, suppose that the architectural template has two different kind of processors: a reconfigurable and a soft one; the application is made of more than one task. Then, one or more VHDL implementations of any task in the task graph might be specified to allow the framework to map that task on hardware (specifically, on the reconfigurable processor). These are called hardware implementations (they might differ for the number of clock cycles required to compute on a bunch of data, and/or the amount of logic required to implement them on FPGA). On the other hand, it is possible to specify a software-based implementation, i.e. a software binary image of the corresponding task compiled for one of the hard or soft processors present in the system. All these different "ways" of realising a computation on the platform are collectively called implementations.

After specifying the implementations, the designer can configure the parameters of the design space exploration phase. These parameters control the behaviour of an underlying meta-heuristic, multi-objective optimisation technique called Ant Colony Optimisation (ACO) [14]. Briefly, ACO realises the design space exploration phase by alternating a local and a global search and iteratively building a complete solution adding more and more choices to an intermediate plan. In the local search phase, an incremental choice is made considering only the choices done so far during the construction of the current solution (metaphorically called "ant"). However, each possible local choice is weighted against a quantity called "pheromone", a weight that summarises the effects of that choice in multiple solutions, which is updated after the end of every iteration on the basis of the performance of the corresponding ant. ACO, with the adoption of adequate heuristics, is an algorithm provably optimum [14].

Lastly, after specifying all the inputs (application, architectural template, implementations and design space exploration parameters) the user invokes the actual exploration tools, collectively called *Design Space Exploration*. DSE realises the cycle in Figure 1: given the inputs, the architectural customisation phase adds or removes components in the predefined areas dedicated to do so (specifically, the reconfigurable area where new components can be instantiated). The exploration never produces unfeasible solutions, i.e. solutions that require too many resources to be implemented in hardware (and specifically, in the reconfigurable area). While this doesn't guarantee that the solution can actually be mapped and routed on hardware, it prevents the computation of solutions that are clearly unfeasible. Moreover, as other state-of-the-art exploration frameworks for MPSoCs [1], [15], [16], A2B's exploration phase realises a simultaneous mapping and scheduling cycle, meaning that the two interdependent decisions (i.e. mapping and scheduling of tasks on an architecture) are taken at once instead of during different decision stages.

The exploration phase tries to optimise a potentially non linear, multi objective metric function using the aforementioned meta heuristic multi objective optimisation algorithm, ACO. The function takes into account multiple metrics and combine them together according to users' needs. While the framework supports metrics such as power and energy consumption, simulated execution time, and area occupancy, we focus only on the minimisation of the execution time of the target application, as we are focusing on the scheduling side of the optimisation problem.

IV. Implementation

In this section, we detail the work-flow of PaRA-Sched. The algorithm is composed of three steps:

1) a *preprocessing step*;
2) the *actual scheduling* phase;
3) the *postprocessing* step.

The first step performs the analysis of task graph and mappings suggested by the exploration algorithm, to add communication and reconfiguration tasks when needed; critical path information are calculated for each task. The second handles the assignment of start and end time estimations to the tasks, while satisfying all precedences between tasks. The last step modifies the task graph introducing additional edges that follow scheduling order.

The work-flow of the scheduler is represented in Figure 2. The following subsections explain in detail each of these steps.

A. Preprocessing step

This step takes as input the original task graph describing the application, and the current mappings computed by the A2B's exploration algorithm; mappings are in form of a vector of tuples $< \text{task}, \text{element}, \text{implementation} >$. Starting from these inputs, the preprocessing phase introduces communications first and then reconfigurations, when needed.

Communication tasks are introduced whenever two processing tasks have a producer-consumer relationship, i.e. when

they are adjacent in the task graph. Two communication tasks are inserted for each edge in the task graph, one *WRITE* task and one *READ* task: the former represents a data transfer of the output of the first task to a memory, the latter represents a transfer from memory to the second task, as input.

Reconfiguration tasks are introduced after communication tasks. The insertion of reconfiguration tasks may not be required in every case, however, it is mandatory to add a reconfiguration if the following condition holds: when two processing tasks must be executed on the same processing element with different (hardware) implementations, from the mapping given by the exploration algorithm, a reconfiguration between them must be performed. Given that the two implementations are different, module reuse is not possible, hence the area must be reconfigured to accommodate the execution of the second task. A reconfiguration task is added in the task graph following this criteria:

- the reconfiguration must be performed after the last *WRITE* that stores in memory data coming from the area to be reconfigured;
- the reconfiguration must be performed before the first *READ* that loads in the area data coming from memory.

The precedences aforementioned are enforced by means of additional edges inserted in the task graph.

At the end of this step, critical path information (ASAP and ALAP estimations and *slack* time), are calculated for each processing, communication and reconfiguration task in the graph. These information are used later as a metric to compute the priority of a task. Dummy start and end nodes are connected to root tasks and final tasks of the graph, the nodes are then ordered topologically and critical path information are computed by means of a *dynamic programming* algorithm.

B. Scheduling

The scheduling phase is performed after the preprocessing and handles the assignment of start/end time estimations to all the tasks in the preprocessed task graph; the assignment must take into account the additional precedences introduced in the preprocessing phase.

The simplified pseudocode of the scheduling phase is outlined in Algorithm 1. The main part of the scheduling phase is composed of a loop, executed after the initialization; each iteration of the loop selects the best task to be scheduled from the list of tasks with precedences satisfied (called *ready tasks*)

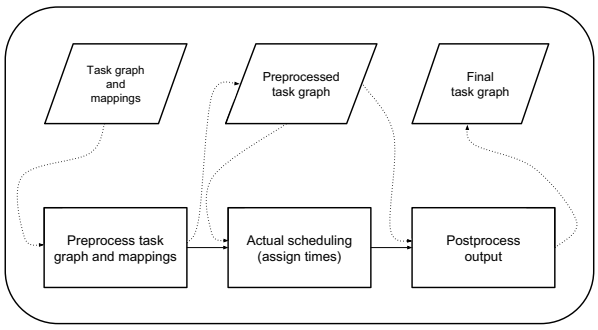
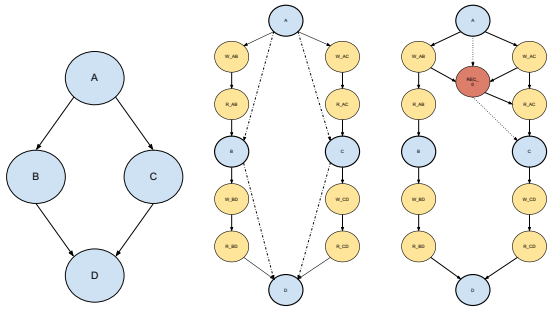


Fig. 2. Work-flow of the scheduler.



(a) Original. (b) With communications. (c) With reconfiguration.

Fig. 3. Evolution of task graph.

Algorithm 1: Scheduling phase

input : the input interface of the scheduler
output: the output interface of the scheduler

```
schedulerOutput ← ∅;
unscheduledSet ← schedulerInput.getTaskSet();
while unscheduledSet is not empty do
    readyTaskSet ← tasks with precedences satisfied;
    bestTask ← best task to be scheduled among the ready tasks;
    timeInfo ← compute start/end time estimations for the best task;
    schedulerOutput ← schedulerOutput ∪ timeInfo;
    readyTaskSet ← readyTaskSet \ bestTask;
    unscheduledSet ← unscheduledSet \ bestTask;
end
return schedulerOutput
```

and assigns time estimations to it, according to the availability of the component on which the task must be executed and the precedence constraints defined by the task graph.

The choice of the best current task is performed by an algorithm, which assigns a priority to each ready task and selects the task with maximum priority value to be returned as best task for the current scheduling step. The priority function is computed based on some metric that depends on the task's type:

- the priority of a processing task with software implementation is only based on critical path information;
- the priority of a processing task with hardware implementation or other types of tasks, i.e. communication or reconfiguration tasks is based on critical path information, earliest start time (EST) and earliest finish time (EFT).

Two additional rules are introduced in the algorithm, to avoid bottlenecks and reduce reconfiguration overhead as much as possible:

- *WRITE* communication tasks that precede a reconfiguration are given an extra priority;
- a decay factor is considered to avoid always choosing

tasks on the critical path as best, a behaviour that might lead to bottlenecks (due to the contention of resources).

The decay factor is initialized to 1; its value remains unchanged until a task on the critical path is chosen to be scheduled. When this happens, the value is decremented by an amount δ_1, so the priority of the next task on critical path is penalized in favor of the other metrics. As soon as the best selected task is not on the critical path, the decay factor is incremented by a certain amount δ_2.

The priority functions are structured in this way (A, B and C are parameters of the scheduler):

$$f_{sw} = \begin{cases} C \cdot CPInfo \cdot df & \text{if task } i \text{ is on the critical path} \\ C \cdot CPInfo & \text{otherwise} \end{cases}$$

for task having a software implementation;

$$f_{hw} = -A \cdot EST - B \cdot EFT + f_{sw} + EP$$

if task i is a ***WRITE*** task preceding a reconfiguration,

$$f_{hw} = -A \cdot EST - B \cdot EFT + f_{sw}$$

otherwise.

After the best task to be scheduled next is chosen, time estimations are assigned by the time assignment algorithm. This algorithm sets the start time estimation of a task to the maximum value between the time at which the component is available and the maximum end time estimation of preceding tasks in the preprocessed task graph. Using a mathematical formulation, the start end time estimation of a task i mapped on a processing element p is computed with the following formula:

$$t_i^s = max(a_p, max(t_j^e)),\ \forall j \in \delta^-(i)$$

where $\delta^-(i)$ is the set of predecessor tasks of task i in the task graph.

The end time estimation is calculated by computing the execution time of the task and adding it to the start time, which means:

$$t_i^e = t_i^s + d_i$$

where d_i is the estimated execution time of task i.

At this point, the best current task has been assigned a feasible estimations that satisfy all precedence constraints and availability of processing elements, hence the new information can be added to the final output and the selected task can be removed both from the ready tasks set and from the unscheduled task set. After the new ready tasks have been computed, a new iteration of the loop begins.

C. Postprocessing

This is the last phase of PaRA-Sched. It is not strictly related to the scheduling of tasks, as all tasks have already been scheduled. The goal of this step is to explicit the scheduling order by introducing new edges in the task graph; the additional edges will enforce an ordering in the execution of all tasks.

In this phase, the (ordered) list of scheduled tasks is retrieved, and all the tasks are connected with edges that follow the order of scheduling. At the end of the process, the preprocessed task graph has been modified and an explicit ordering has been set by the newly introduced arcs.

V. Evaluation Results

A. Evaluation Platform

We implemented PaRA-Sched and the exploration algorithm ACO-based in C++, and evaluated the algorithm by means of several synthetic task graphs. The benchmark used for the tests have been generated with TGFF, as in [17]. The platform used for the generation and evaluation of the experimental results is composed of an Intel® i7 processor with 4 cores and 2.8 GHz of clock frequency.

B. Experimental Results

The generated task graphs have a number of nodes that ranging 10 to 70, to simulate the execution with very small applications or even multiple applications executed at the same time on the target device, in case of large task graphs.

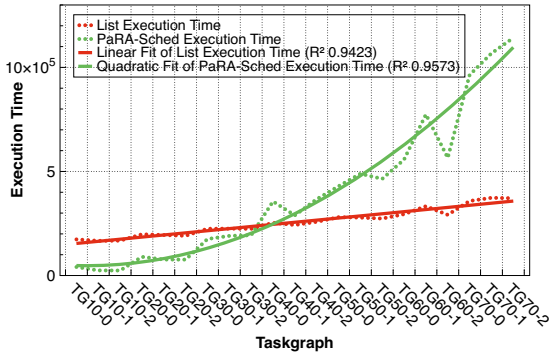
The benchmarks executed involve three different architectural templates: *static* architectures have a reconfigurable area on the FPGA partitioned in a set of k_S static cores (IP cores); *mixed* architectures feature k_M^S reconfigurable cores and k_M^R static cores on the FPGA area; finally, *reconfigurable* architectures are composed only of k_R reconfigurable regions.

For each architectural template, two different architecture have been generated, based on Xilinx Zynq-7000 FPGA: an Artix-7 and a Kintex-7. The first has a reconfigurable area with $28,000$ logic cells, whereas the second has $125,000$ logic cells.

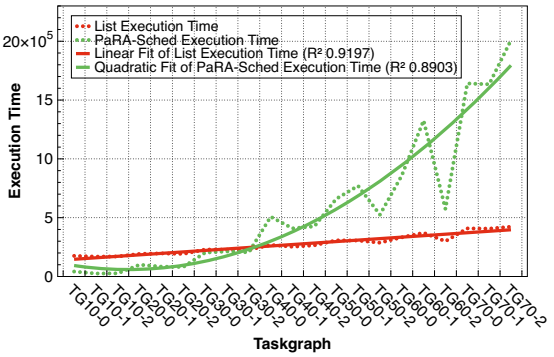
All the results reported in this section have been generated by executing the ACO-based exploration with 70 iteration each of which has 5 ants. The size of task graphs using for the tests ranges from 10 to 70 nodes.

Figure 4 depicts and compares the execution times of PaRA-Sched with respect to list-based scheduler. These results have been collected during the execution of the test used to retrieve makespan values of both schedulers; on the Y-axis is represented in microseconds the execution time of all the invocations of the scheduler during the exploration. Results obtained show that the execution times of the list-based scheduling algorithm are lower than those of PaRA-Sched. This behaviour is expected, since the reasoning performed by the scheduler impacts on the performance of the algorithm and reduces its speed, while improving the quality of the solutions generated. It is worth noting that the execution time of PaRA-Sched follows a quadratic curve, except for some task graphs; for example, the instance TG060-2 is solved in less time than predicted using a quadratic fit function. This could be produced by that particular instance of the problem being particularly easy to deal with in the scheduling phase.

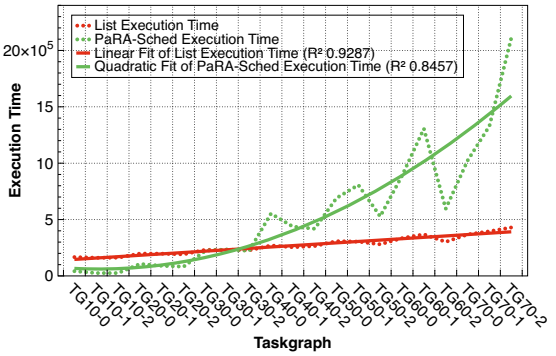
Figure 5 represents the makespans returned by the list-based baseline algorithm compared to those computed by PaRA-Sched; on the Y-axis is represented the lowest possible makespan found during the exploration phase for each task graph. Results show that PaRA-Sched is capable to generate



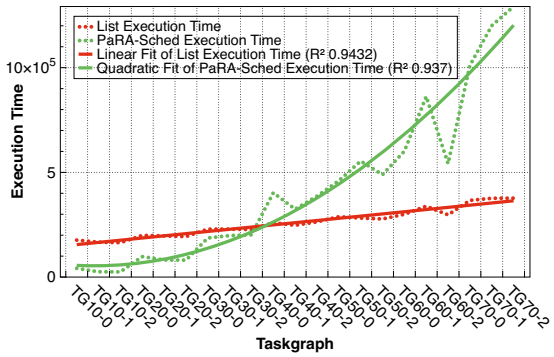
(a) Baseline architecture with two software processors.



(b) Mixed architecture with two software processors, 15 IP cores and 15 reconfigurable regions.



(c) Reconfigurable architecture with two software processors and 30 reconfigurable regions.



(d) Static architecture with two software processors and 30 IP cores.

Fig. 4. Comparison of execution times. Y axis is in microseconds and X axis is the ID of the task graph under examination, ordered by increasing number of task nodes (from 10 to 70). All results have been generated using a Kintex-7 architecture; linear and quadratic fits are provided for the list-based scheduler and PaRA-Sched, respectively, to empirically show the model that best approximates the execution times.

better schedule with respect to the list-based scheduler, because of the heuristic applied to select which task is the best candidate to be scheduled. It is worth noting that the bigger the processed task graph is, the better PaRA-Sched performs with respect to list-based scheduling. In particular, when the architecture is composed by a small number of processing elements, the difference of the makespan is more emphasised, especially in case of static design, when contention on the soft-core processor becomes a drawback and the correct scheduling of processing tasks impacts significantly on the performances. Also in the case of reconfigurable design with a small number of reconfigurable cores the difference in the performance is noticeable. One possible reason behind this behaviour is that communications become a critical factor in a large task graph with many reconfigurations that have to be performed, and an incorrect scheduling of them can lead to a huge overhead and penalisation of the makespan. By adopting techniques such as the extra priority assigned in case of communication tasks that precede a reconfiguration, the overall makespan can benefit.

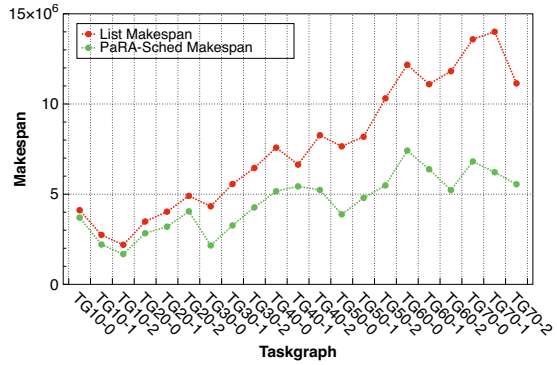
An important remark concerns the exploitation of partial dynamic reconfiguration in the design process of application targeted to reconfigurable devices. As shown in Figure 5 the employment of partial dynamic reconfiguration by creating reconfigurable regions allows to keep the overall makespan of reconfigurable and mixed designs in line with static ones, while at the same time preserving the amount of resources employed to do so. This is fundamentally due to module reuse and reconfiguration time masking by means of configuration prefetching.

VI. Conclusions and Future Work

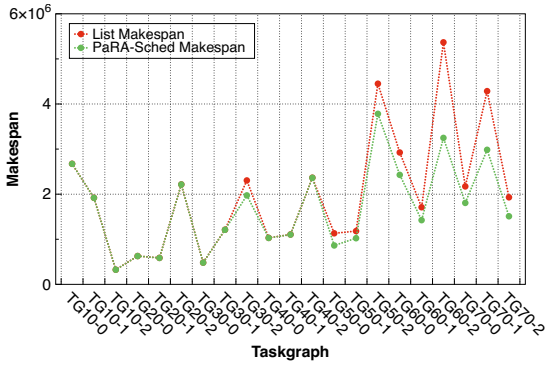
We presented PaRA-Sched, a novel scheduling algorithm for the design space exploration of FPGA-based reconfigurable systems. We introduced the explicit notion of PDR in the scheduling infrastructure, and demonstrated that the methodology can effectively exploit this feature to improve the performance of all the corresponding static solutions that do not make use of this feature. Future work will further investigate the impact of PDR on said solutions and improve the algorithms with further heuristics.

Acknowledgments

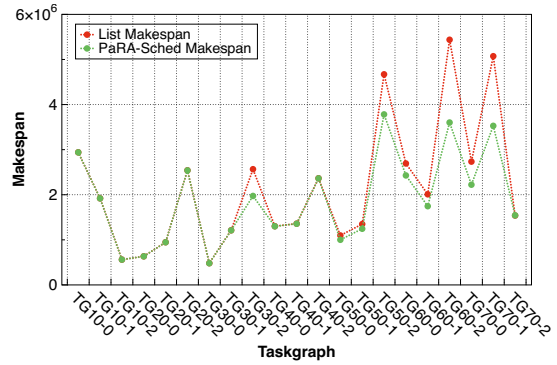
This work was partially funded by the European Commission in the context of the FP7 FASTER project (#287804).



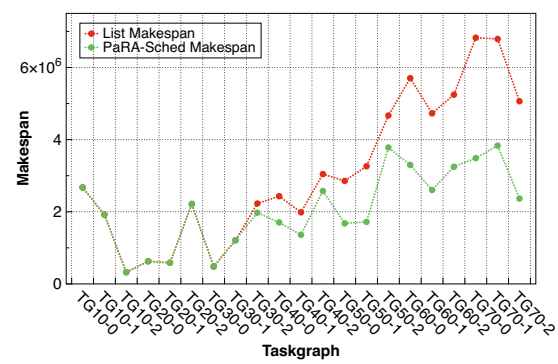
(a) Baseline architecture with two software processors.



(b) Mixed architecture with two software processors, 15 IP cores and 15 reconfigurable regions.



(c) Reconfigurable architecture with two software processors and 30 reconfigurable regions.



(d) Static architecture with two software processors and 30 IP cores.

Fig. 5. Comparison of makespans. Marked dots represent the makespan computed by the two scheduling algorithms.

REFERENCES

[1] M. Santambrogio and D. Sciuto, "Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign," in *Proceedings of IPDPS '08*, 2008, pp. 1–8.

[2] A. Sangiovanni-Vincentelli, L. Carloni, F. D. Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of DAC '04*, 2004, pp. 409–414.

[3] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Integrating Physical Constraints in HW-SW Partitioning for Architectures With Partial Dynamic Reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, 2006.

[4] F. Ferrandi, P. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 6, pp. 911–924, june 2010.

[5] C. Pilato, R. Cattaneo, G. Durelli, A. Nacci, M. Santambrogio, and D. Sciuto, "A2b: An integrated framework for designing heterogeneous and reconfigurable systems," in *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, 2013, pp. 198–205.

[6] 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, 2006.

[7] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch, "Resource-constrained project scheduling: Notation, classification, models, and methods," *European Journal of Operational Research*, vol. 112, no. 1, pp. 3–41, Jan. 1999. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0377221798002045

[8] 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.

[9] 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. Comp.*, vol. 2008, 2008.

[10] *A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems* . ProRisc Workshop CKTS, 2000.

[11] C. Fiduccia and R. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Design Automation, 1982. 19th Conference on*, pp. 175–181, June 1982. [Online]. Available: http://www.bibsonomy.org/bibtex/2f310eb1147cd6e29c5cd055ec3d15f9e/lee_peck

[12] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell Systems Technical Journal*, vol. 49, no. 2, 1970. [Online]. Available: http://www.bibsonomy.org/bibtex/29775c75c01f914a382cf65ec8791d4cc/wvdaalst

[13] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures." in *ASP-DAC*, F. Hirose, Ed. IEEE, 2006, pp. 491–496.

[14] M. D., Middendorf, M., and H. Schmeck, "Ant colony optimization for resource-constrained project scheduling," in *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 4, 2002, pp. 333–346.

[15] D. Gohringer, M. Hubner, M. Benz, and J. Becker, "A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip," in *Proceedings of FCCM '10*, may 2010, pp. 259 –262.

[16] J. Clemente, V. Rana, D. Sciuto, I. Beretta, and D. Atienza, "A hybrid mapping-scheduling technique for dynamically reconfigurable hardware," in *in Proceedings of FPL '11*, 2011, pp. 177–180.

[17] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 6, pp. 911–924, 2010.