SMASH: A Heuristic Methodology for Designing Partially Reconfigurable MPSoCs

Riccardo Cattaneo, Christian Pilato, Gianluca Durelli,
Marco D. Santambrogio, Donatella Sciuto
Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
{rcattaneo,pilato,durelli}@elet.polimi.it, {marco.santambrogio,donatella.sciuto}@polimi.it

***Abstract*—The exploitation of the capabilities offered by reconfigurable architectures is traditionally a demanding task due to the intrinsic time consuming and error prone customization of these systems around the specific application. Moreover, existing approaches are not able to integrate the notion of partial and dynamic reconfiguration (PDR) from the early stages of the decision phases, potentially leading to sub-optimal solutions.**

In this work, we propose *SMASH* (Simultaneous Mapping and Scheduling with Heuristics), a highly automated design methodology focused on explicitly taking into account PDR during the design of reconfigurable designs. It combines heuristics for both the design of the architecture and the mapping and scheduling of the partitioned application. We show how this additional degree of freedom leads to architectures whose performance are improved with respect to the baseline.

I. Introduction

Nowadays, the design of efficient embedded systems relies on heterogeneous MPSoCs [1] that combine general purpose processors with dedicated hardware accelerators. Indeed, efficient tools (e.g., Xilinx Vivado HLS, Synopsys C Compiler, Cadence C-to-Silicon, Calypto CatapultC) are becoming very popular for the automatic generation of hardware implementations from the corresponding behavioral specifications, also allowing the possibility to explore different implementations (e.g., trade-offs between performance and requirements of resources). As a result, high-performance and low-power architectures can be obtained with the hardware acceleration of different parts of the application [2], even if their design still requires high expertise.

In particular, Field Programmable Gate Arrays (FPGAs) are very attractive solutions that allow implementing large parts of the application in hardware at low cost. Moreover, exploiting *Partial Dynamic Reconfiguration* [3] (PDR) offers the possibility of reusing some part of the logic across different tasks, despite of an overhead in the execution time required to reconfigure the corresponding logic cells. For this reason, this technique introduces several challenges that have to be properly taken into account during the design of such systems. In particular, one of the main issues in designing heterogeneous systems, especially when PDR is taken in account, is the customization of the architecture [4] in terms of hardware accelerators (either static or reconfigurable) that are usually defined in advance, potentially leading to sub-optimal solutions. Then, the designer has to determine which tasks have to be hardware accelerated and the level of reconfiguration for each of them (if any), also with respect to the number of available resources. Moreover, since a task reconfiguration requires to load the new configuration bitstream for the corresponding region, this can introduce a penalty in the execution time if it is not properly taken into account in the design of the application [5]. Finally, the effects of the data transfers between the tasks are crucial aspects [6] and thus the impact of the interconnection infrastructure (e.g., bus, NoC, FIFO) has to be necessarily taken into account. In conclusion, novel and efficient methodologies are definitely required to take into account the reconfiguration aspects for the early stages of the design process.

In this paper, we propose *SMASH* (Simultaneous Mapping and Scheduling with Heuristics), a design methodology that aims at addressing the limitations cited above. It combines different heuristics for both customizing the architecture and implementing the application to generate reconfigurable systems tailored for the input partitioned specification. Indeed, it determines which implementation has to be adopted for each task and the level of reconfiguration of the corresponding hardware modules, also potentially taking into account different topologies that can adopted for interconnecting the processing elements. In fact, SMASH includes an exploration phase that is able to determine the proper mapping and scheduling for the different tasks of the application, determining the tasks' implementations and the processing elements where they have to be executed. During this exploration, it also determines which hardware modules have to be included into the reconfigurable logic (either to be used as static or reconfigurable regions) and the resulting reconfigurations are taken into account during the evaluation of the solution, along with the required communications. Moreover, taking into account the resulting resource requirement of the different regions during the exploration allows to limit the generation of unfeasible solutions. Simulations by means of virtual platforms have been adopted to validate the proposed approach.

The rest of the paper continues as follows. Section II overviews existing approaches that aim at addressing similar problems, highlighting the contributions of the proposed solution whose overall organization is presented in Section III. Then, Section IV and Section V present the heuristics at the basis of this work and the solution evaluation method, respectively. In Section VI presents the experimental evaluation of the proposed approach, while Section VII concludes the paper and outlines the future directions of work.

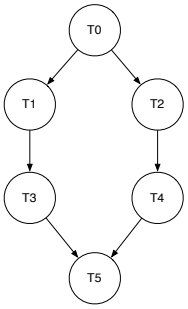

Fig. 1: Example of application DAG.

II. Related Work

The synthesis of heterogeneous MPSoCs usually requires an efficient exploration of the design space. Daedalus [7] is an interesting and integrated framework for starting from a sequential application and then generating the corresponding parallel implementation, along with the corresponding system. However, it only focuses on streaming applications and reconfiguration aspects are not taken into account. It is worth noting that these aspects are usually not taken into account for such applications due to synchronization issues in the FIFOs when reconfiguring the blocks. Moreover, single stages of this computation are usually quite simple and, thus, reconfiguration is usually not attractive in this scenario. On the other hand, task-based applications are usually characterized by time-consuming computational blocks interleaved by data transfers. They are usually represented as Direct Acyclic Graphs (DAGs), as the example shown in Figure 1.

As opposite to creating full-custom architectures, platform-based design [4] is instead a viable solution for reducing the complexity of designing such systems by a a progressive refinement of an architectural template. Based on this idea, different approaches [6], [8], [9] have been proposed for optimizing partitioned applications, especially in the case of hardware acceleration. In particular, [8] uses an approach based on task clustering, while [6] explores different mapping and scheduling alternatives with a constructive approach for limiting unfeasible solutions. However, in both of the cases, PDR is not addressed and thus tasks can be executed in hardware as long as they fit in the available area. On the other hand, [9] proposes a method for mapping and scheduling of reconfigurable systems, but the target architecture (e.g., the number of reconfigurable regions) have to be defined in advance, potentially leading to sub-optimal solutions.

In [10], the authors propose a set of techniques focusing on the partitioning of the code and the generation of the corresponding adaptive system. However, they mainly focus on dynamic aspects and the support for the Operating System (OS), while we are more interested in design-time decisions, in order to have a very lightweight OS or even a bare-metal synchronization of the application.

On the other hand, it is worth noting that, for creating a feasible implementation of the system, another step is usually required: the definition of the physical constraints within the FPGA to satisfy the resource requirements (e.g., LUTs, BRAMs, DSPs) of the hardware modules and to avoid their overlapping. The authors in [5] propose a methodology for mapping the tasks to processing elements taking into account reconfiguration aspects and also placement issues. However, their assumptions are quite simplistic (e.g., homogeneous resources for the regions) and they are difficult to be applied to recent FPGA devices, where the designer can design very different and two-dimensional reconfigurable regions. In this paper, we separate the problems: we identify the number of hardware modules while exploring the mapping of tasks with respect to them. Then, we only verify that the total amount of resources required by the regions can be effectively satisfied by the target FPGA. Extending the proposed approach to integrate the verification of the physical constraints is straightforward: a floorplanning algorithm (e.g., the one proposed in [11]) can be integrated in the evaluation of the solution and return to the exploration algorithm if the assignment results in a feasible allocation of the resulting regions or not. However, this is out of the scope of this paper and it has been left as a future work.

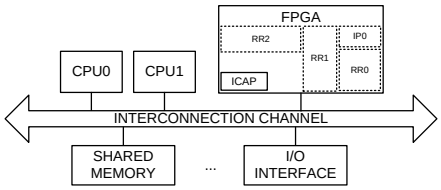

Fig. 2: Example of target architecture.

In conclusion, the main contributions of the proposed approach can be summarized as follows:

- it optimizes the execution of the given task graphs with respect to an architectural template, determining the level of reconfiguration for each of the tasks that are decided to be executed in hardware and the nature of the hardware modules to be introduced in the final platform;
- it defines an exploration framework that can easily accommodate different algorithms, metrics and evaluation methods to design a reconfigurable system;
- it generates the specification of the virtual platform corresponding to the identified solution.

This approach has been validated by means of synthetic applications that are representative of real-life applications. The corresponding solutions have been then evaluated with high-level simulations of the generated virtual platforms through Synopsys Platform Architect [12].

III. Proposed Methodology

The proposed methodology, namely SMASH, starts from the description of the architectural template to be customized (as the one shown in Figure 2) and of one or more partitioned applications to be concurrently executed. Each of these applications can be represented as a DAG (as the one shown in Figure 1) and a unique representation can always be obtained by combining them. A list of admissible implementations (e.g., combination of performance and resource requirements) has to be also provided for each task. For software tasks, they can be computed by profiling or estimating its execution. On the other hand, for hardware tasks, it is possible to obtain both execution time and required resources by estimations or by actual synthesis through a HLS tool such as Vivado HLS [13].

SMASH is then applied to this resulting DFG and it is mainly composed of two parts, as shown in Figure 3:

1) an exploration of the mapping and scheduling for the input DFG to statically determine which implementation has to be adopted for each of the tasks and where they have to be executed (e.g., processors or reconfigurable logic).
2) a final customization of the architecture, where it is possible to identify static IP cores (i.e., modules with only one task associated with), as well as reconfigurable regions.

As output, it produces a description of the system to be implemented, including the specification of the customized architecture and the mapping and scheduling of the tasks (including reconfiguration ones) with respect to this generated architectural solution.

In details, the exploration heuristic (further detailed in Section IV) aims at evaluating different solutions in terms of mapping and scheduling to determine the best implementation for the given DFG with respect to the target architecture enhanced with hardware modules. In particular, when assigning multiple tasks to the same region, the heuristic is able to automatically compute its overall resources requirements and, taking into account the requirements of all the blocks, whether the solution is feasible or not. Then, the solution evaluation determines the reconfiguration tasks that have to be introduced (i.e., when consecutive tasks executing different functions are assigned to the same module) and evaluates the performance of the solution taking into account also the reconfiguration overhead. Note that this term is computed on the basis of the size of the reconfigurable region (i.e., its requirement of resources) as it results from the generated mapping.

The last phase of the methodology is a post-processing step that analyzes the mapping solution and the architectural instance generated after the first phase. In such a situation, each hardware module that has been introduced by the exploration algorithm can have one or more tasks assigned. If the module has only one task assigned, it means that it can be converted into a static IP block, since no reconfiguration is required, thus reducing its area consumption. Otherwise, modules with more than one task assigned are represented as actual reconfigurable regions in the final architecture.

In conclusion, the methodology can produce the description of the resulting system. In particular, we assume that the generated solution performs correctly from the functional point of view and then we are only interested into evaluating non-functional properties of the system (e.g., performance). For this reason, in this work, we generated a virtual platform for the high-level evaluation of the solutions, where each processing element is represented as a Virtual Processing Unit (VPU) and all the VPUs are interconnected as specified by the input architectural template. Then, the initial DAG is mapped onto these VPUs as specified by the generated mapping solution and the reconfiguration tasks are assigned to the VPU that has in charge of performing the actual reconfiguration (e.g., a dedicated processor or one of the available GPPs) to correctly model its execution overhead.

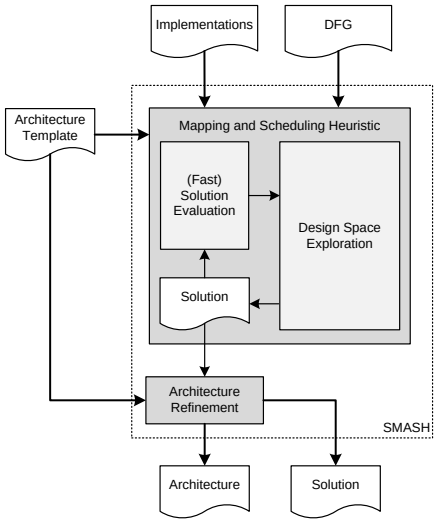

Fig. 3: Overview of the proposed methodology.

IV. Mapping and Scheduling Exploration

Our mapping and scheduling exploration is based on Ant Colony Optimization (ACO) [14]. This meta-heuristic relies on the abstraction of an agent (i.e., the "ant") stochastically exploring the design space, described as a sequence of choices. At each step, the agent computes all the available choices and ranks them according to a rule which helps the algorithm finding the best solutions, iteration by iteration.

The rule for ranking choices relies on two heuristics: *local* and *global*. The former is a function that assigns a score to a choice given the current state of the exploration, and integrates knowledge about the specific problem. It allows the agent to make an informed (yet local) decision about the next step of the exploration. The latter is a model for the abstraction of ants' *pheromones*, a substance ants naturally release on the path they traverse and to which are naturally attracted. Being a volatile substance, the intensity of the pheromone trail will eventually disappear at a rate inversely proportional to the number of ants traversing (i.e. reinforcing) it. On the other hand, on a heavily traversed path the deposited pheromone trail will be reinforced, and more ants will be attracted to it. If a path to a target is optimal, ants will reach it in lesser time than other paths; thus, the amount of pheromones on that trail will evaporate relatively slower because it takes less time to traverse it (in other words, it is reinforced more frequently). Since the intensity on that trail is relatively higher, relatively more ants will be attracted to it, releasing themselves more pheromones thus reinforcing again the trail. In a sense, the amount of pheromones on a path globally keep track of how good that path is to reach the goal.

An ACO algorithm implements the exploration in an iterative way by using "generations of ants". A *generation* is composed of N ants performing design space exploration (DSE), all using the same values of the global pheromone matrix. At the end of each generation, only some ants (i.e., a parameter K defined by the user) are considered for global heuristic reinforcement, proportional to the quality of the corresponding solution. After a convergence criteria is met, the best solutions are kept and the others are discarded.

A. Detailed overview of the algorithm

This algorithm describes the steps required to map an application represented as a DAG onto a reconfigurable architecture composed of different processing elements (i.e., software processors, static IP cores and reconfigurable regions) using the ACO technique with the goal of minimizing the total execution time.

Algorithm 1: Overview of exploration algorithm

```
   input : A task graph and a reconfigurable architecture
   output: A mapping trace
1  forall the numGenerations do
2      forall the antsPerGeneration do
3          readySet ← tasksWithoutPreds ()
4          scheduledSet ← ∅
5          while readySet ≠ ∅ do
6              forall the T_i ∈ readySet do
7                  p_{T_i} ← localHeuristic (T_i)
8              chosenT ← roulette (p_T)
9              iSet ← implsOfTask (chosenTask)
10             forall the I_i ∈ iSet do
11                 pSet ← processorsPerImpl (I_i)
12                 forall the P_i ∈ pSet do
13                     p_{I_i,P_i} ← localHeuristic (I_i, P_i)
14             chosenI, chosenP ← roulette (p_{I,P})
15             choice ← <chosenT, chosenI, chosenP >
16             mappingTrace.add (choice)
17             readySet ← resolveDependencies ()
18         ant.Metrics ← computeMetrics (currentAnt)
19         ant.Objective ← computeObjective (ant.Metrics)
20         thisGenerationSolutions.add (ant)
21     bestAnts.add (selectBest (thisGenerationSolutions))
22     updateGlobalPheromones (bestAnts)
23 bestAnt = selectSingleBestAnt (bestAnts)
24 return bestAnt.trace
```

The algorithm evolves a certain number of generations of ants (lines **1**-**22**) until termination criteria is met. In our case, the termination criteria is the number of generations to evolve. Each ant builds a solution as a sequence of *choices* (lines **2**-**22**): in our specific case, they are mapping choices, i.e., 3-tuples of the form $< task, implementation, processing\_element >$ representing how a task should be implemented and where. In this context, an implementation is one of the available ways to execute a task either in software or in hardware. An ant computes a complete *trace*, i.e. a list of mapping choices where an explicit scheduling priority is expressed by the ordering of the choices in the list itself.

In order to iteratively build a solution, the ants generate all feasible choices at each step of the algorithm, ranking each of them according to a rule based on two heuristic functions, one for the choice of the task to map (lines **6**-**8**) and one to choose both the implementation and the processing element to execute this task onto (lines **10**-**14**). Note that the exploration starts with a minimum number (min_{HW}) of hardware modules to employ into the final architecture. Then, at each step, the choice may reuse the allocated modules or may instantiate an additional hardware module in order to better exploit hardware resources, given that the area constraint is never violated and the total number of hardware modules is less than a maximum value (max_{HW}) or the number of tasks. After assigning a score to each mapping choice (line **13**), the ant selects one of those choices using a roulette wheel selection scheme (line **14**), where the probability of each choice is proportional to its heuristic value. After a decision for this iteration is made, the ant computes which tasks may be executed afterward (line **17**), and the process continues until no more tasks need to be scheduled (lines **5**-**17**).

At this point, the solution is evaluated according to a reconfiguration-aware scheduler (further detailed in Section V), and an objective function is assigned to it (line **19**). When the last ant of the current generation ends, the best solutions are used to reinforce the global pheromone matrices (line **22**).

The algorithm is known to efficiently solve large instances of task mapping problems [6]. However, we further improved it to devise efficient schedules exploiting knowledge about the problem and the reconfigurable architecture onto which tasks are mapped.

B. Reconfiguration-aware Heuristics

As described in IV-A, a 2-step decision process is adopted to reduce the complexity of the exploration, as in [6]. The first one is the *task* local heuristic, which selects, at each decision point, what is the current "best" task to schedule among the set of ready tasks. The second one is the *mapping* local heuristic, which selects, at each decision point, given a task, what is the current "best" choice of processing element and implementation to execute the chosen task with. To balance the relative weight of all the choices, all heuristic values (local and global) are scaled in the $(0, 1)$ interval, where higher means better choices.

The *task* local heuristic yields high values for tasks characterized by low mobility values and relatively fast execution times (averaged over all the task's possible implementations). This rule particularly favors an early execution of ready tasks on the critical path that might otherwise unnecessarily increase the overall execution time. It also slightly increases the heuristic value for tasks that executes relatively faster than the others.

The *mapping* local heuristic, instead, is computed in differing ways depending on the implementation. Moreover, since the architecture might feature static, reconfigurable and software processors, we have to rank the choices featuring any of them in a way that they can be fairly compared to each other. The first step, then, is to understand whether the implementation is hardware or software.

If software, we compute the heuristic as the product of two terms in the $(0, 1)$ interval: H^1_{SW} and H^2_{SW}. H^1_{SW} is the likelihood of a software implementation of the task with respect to *any* hardware one (if available), to take into account how a software implementation might be better or worse than a hardware one. H^2_{SW} computes the *average mobility* of all software processors, as the sum of the mobility values of all the tasks previously assigned to each software processor, divided by the number of those tasks. Software processors that haven't been assigned tasks yet are automatically ranked

better than any other software processor choices. This choice privileges software processors that have never been assigned tasks before and that have been previously mapped with tasks with relatively lower average mobility.

If hardware, instead, we consider the set of static IP processors and reconfigurable regions available in the system. For the computation of the heuristic of a reconfigurable region, we compute the product of three terms in the $(0, 1)$ interval: H^1_{HW}, H^2_{HW} and H^3_{HW}. H^1_{HW} is a general likelihood factor for a hardware implementation with respect to *any* software one (if available). H^2_{HW} is a factor that relates to area usage for instantiating the implementation on this reconfigurable region. This term takes into account the number of tasks that are yet to be scheduled in order to assign each task a (potentially) fair share of the resources of the FPGA. We penalize the heuristic values of those choices which would consume more resources (in percent terms) than the percent advancement of the mapping algorithm. For example, if the ant is mapping the 5th task out of 10 available tasks, a good choice must not consume more than 50% of the available FPGA area, after mapping this task to the choice's reconfigurable region. If this limit is violated, but the overall constraint on total FPGA area is not violated, the choice is viable but is consuming more area than it should and so its heuristic value is halved. If instead the violation leads to a global area violation, the solution is discarded (i.e., heuristic value = 0, preventing to take this decision). This allow for an early discovery of unfeasible solutions for faster convergence. H^3_{HW} term is similar to H^2_{SW}: the lower the average mobility of the tasks already assigned to a reconfigurable region, the better this term.

V. Solution Evaluation

At each iteration, it is required to evaluate the current solution and provide feedback to the exploration heuristic. In this paper, we only considered the overall execution time of the application as the metric to be optimized. However, the framework has been designed to accommodate the usage of different metrics (e.g., execution time, power consumption and area occupation) or any combination of them. Furthermore, it is possible to integrate different methods to compute the metrics, ranging from mathematical models to actual simulations, to trade-off elaboration time and accuracy of the evaluation. As an example, the designer can adopt cycle-accurate simulations when analyzing small applications and then simpler heuristics when considering large ones. Also the method adopted for the final validation, i.e., high-level simulation with virtual platforms, can be integrated. In all the cases, the metrics are intended to work on scheduling trace of the DAG produced by the exploration heuristic previously described.

It is worth noting that the scheduling trace only contains the mapping decisions for the tasks on the processing elements; however, given the possibility of exploiting PDR, it is clear that this trace is not enough to determine the application execution time since it does not include the reconfiguration tasks. For this reason, it is then necessary to construct a more detailed representation of the application to be evaluated, starting from this scheduling trace. Note that the transformations that we applied to construct this enhanced representation can be applied to compute any of the metrics mentioned above; however the designer can also apply other transformations to support even more detailed descriptions of the system.

TABLE I: Example of scheduling trace for the TG in Fig. 1

Task Name	Implementation	Processing Element
T0	A	CPU0
T1	A	RR0
T2	B	RR0
T3	A	RR1
T4	A	RR0
T5	C	CPU0

As an example, we consider the trace shown in Table I: it reports an admissible mapping and scheduling for the DAG of Figure 1 with respect to the architecture shown in Figure 2. It is worth noting that reconfigurations are extracted only when two consecutive tasks are assigned to the same region, but with different implementations. This allows supporting hardware reuse, when multiple tasks are assigned to the same region with the same implementation; in fact, in this case, reconfigurations are not required to switch the functionality.

Starting from this trace, we introduce two more entities in the graph to be scheduled which are the *reconfiguration nodes* and the *communication nodes*. The former represents a reconfiguration that has to be performed to change the functionality of a hardware module and it is mapped on a dedicated component (i.e., the ICAP on Xilinx devices). The latter, instead, is used to represent the data exchanged directly between tasks or through the memories, according to the communication infrastructure. Given this enhanced representation, we adopted a simple yet effective list-based scheduling algorithm [15] to compute the overall execution time (i.e., the make-span) of the application. The execution time of each task is reported into the implementation determined by the mapping, while reconfiguration and communication overheads are computed as described below.

A. Reconfiguration nodes

The heuristic described in Section IV assumes the possibility that the hardware modules may have multiple tasks assigned to the same hardware module. Adding reconfiguration nodes consists in identifying where the reconfigurations take place and performing an estimate of their execution time. Starting from the information in Table I, it is possible to determine where reconfigurations occur by identifying which hardware modules implement more than one different implementation. In this example, it is possible to identify reconfigurations only for the module *RR0*, since *RR1* is set to execute only *T3*. Once the reconfigurations have been identified, they have to be accordingly introduced in the task graph. In particular, each reconfiguration is inserted between the task implemented on the hardware core and the one that needs to be reconfigured for the execution on the same module. Furthermore since the scheduling trace also defines the priorities between the tasks, the reconfigurations must respect this order and they can be accordingly ordered. Given the trace in Table I, the reconfiguration nodes identified are reported in Table II and the resulting task graph is reported in Figure 4 (left).

TABLE II: Reconfiguration nodes identified starting from the trace reported in Table I.

Rec. Node	Proc. Elem.	Function	Prev. Task	Next Task	Prev. Rec.
REC0	RR0	B	T1	T2	-
REC1	RR0	A	T2	T4	REC0

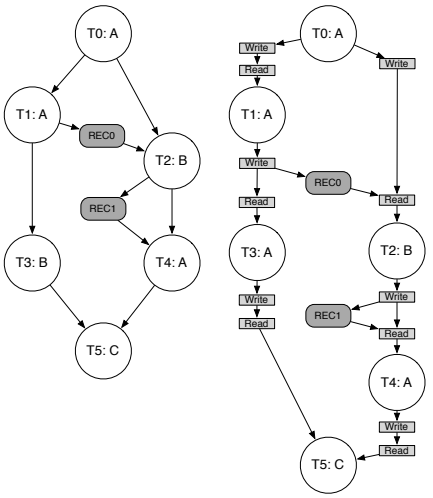

Fig. 4: Task Graph of Figure 1 extended with reconfiguration nodes (on the left) and with both reconfiguration and communication nodes (on the right).

Concerning the execution time of a reconfiguration task, we assume that it is proportional to the size of the largest bitstreams of the implementations assigned to the corresponding hardware module. Then, we adopt the same approach proposed in [16] to estimate the actual reconfiguration time required for each region. In our algorithm all the reconfiguration nodes are scheduled to be executed in sequence by a dedicated processor.

B. Communication nodes

After the reconfiguration, also the communication nodes can be introduced, based on the topology of the interconnection between the modules where the tasks have been assigned. Representing communications as explicit nodes and keeping them separated from the actual computation of the tasks allows to provide, if needed, a more accurate simulation of the overall system behavior. For example, it is possible to include bus congestion metrics during the simulations and envision exploration phases that aim at customizing also the communication architecture for the given application. Considering a bus-based architecture with shared memory, before executing a task, it is necessary to read data from the memory and, after its termination, to send the results to the memory to be used by subsequent tasks. Thus, communication nodes are added before (*read*) and after (*write*) each task and their execution time is proportional to the amount of data to be transferred. The result of integrating communication nodes to the task graph of Figure 1 is reported in Figure 4 (right).

Regarding the communication time, this is estimated based on the given amount of data to be transferred, as specified in the DAG representation. Note that exploration of data transfers can be also integrated, as in [6], but this is out of the scope of this work.

VI. Experimental Evaluation

We implemented SMASH in C++ and we then applied the resulting framework to a set of benchmarks generated with TGFF, as in [6]. We then generated the Virtual Platforms (VPs) corresponding to the resulting solutions and we simulated them with Synopsys Platform Architect [12] for rapid prototyping of system-level integration. This allows us to validate the proposed approach in a wide range of case studies and also to analyze the scalability of the approach. In particular, the generated task graphs are converted to VP models by using the *Generic Task Library*, where their *processing time* is based on the implementation details which the corresponding tasks have been mapped onto. Then, each processing element model is typically available in the Platform Architect library as a Virtual Processing Unit (VPU). VPUs can thus represent all programmable, configurable, or fixed logic processing elements, based on the specified VPU configurations. Also the ICAP controller of the reconfiguration process is modeled as a VPU to execute reconfiguration tasks. Then, modules of other regular IP blocks (e.g., interconnect, memories and DMAs) are instantiated to allow a complete simulation.

The generated task graphs ranges from 10 to 100 nodes, where each task has at least one SW implementation and multiple HW ones, representing realistic trade-offs between execution time and requirement of resources. Large task graphs can also represent multiple applications to be simultaneously executed onto the target platform. The adopted architectural templates (similar to the ones in Figure 2) can easily represent embedded systems featuring either soft or hard processors (e.g., Xilinx XUPV5 with Microblazes [17] or the newer AVNET Zedboard with ARM Dual Core Cortex-A9 [18]), augmented with a set of hardware modules. We adopted three architectural templates as starting point for our experiments: *static*, *mixed* and *reconfigurable*. *Static* identifies architectures where the FPGA area is divided into a set of up to k_S static IP cores. *Mixed* identifies architectures where both IP cores and reconfigurable regions are employed to devise a solution, but no more than k_M^{IP} IPs and k_M^R reconfigurable regions may be used at once. *Reconfigurable* represents an architecture with no more than k_R regions. Note that in the second and the last cases, the reconfigurable regions can be also deployed as static cores in the final architecture, in case they are assigned with only one task. We generated two sets of architectures from these templates, based on commercially available Xilinx Zynq-7000 FPGA devices: an Artix-7 and a Kintex-7, with $28,000$ and $125,000$ logic cells, respectively. For the lack of space, we report only the results related to Kintex-7, given that the other ones show a similar behavior.

Figure 5 reports the results obtained when executing SMASH for 75 generations with 10 ants for each of them. Indeed, this combination of parameters is an empiric good compromise between greedy search and evolution towards a global optimum based on the pheromone matrices. Note that, in the graph, we reported the execution times normalized with respect to the fully static execution. Table III reports instead information about the resulting architectures, along with information about hardware accelerated tasks and reconfigurations.

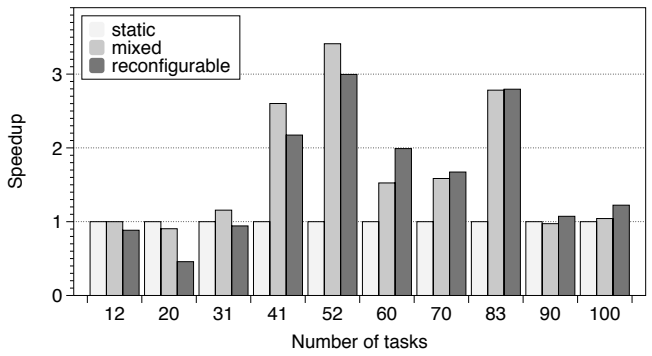

Fig. 5: Speedups of mixed and reconfigurable architectures with respect to the static one.

Results show that SMASH is always able to map a large number of tasks in hardware, generally obtaining relevant speedups when starting from mixed and fully reconfigurable templates. In particular, between 40 to 80 nodes, SMASH is able to obtain speedups up to 3x with respect to static architectures, which limits the number of tasks that can be ported in hardware without exploiting PDR. It is worth noting that, for small instances, reconfiguration can also introduce a slow-down in the execution time since, in this case, its overhead has a proportionally larger impact. On the other hand, larger instances show limited speedups because less tasks can fit in the available area and both software tasks and data transfers can affect the execution time.

The results in Table III also show that SMASH is effectively able to identify the best combination of static IP cores and reconfigurable regions based on the problem structure. Indeed, when starting from both mixed and fully reconfigurable architectures, it is able to devise which hardware modules require to be reconfigured and, at the same time, determine how to assign and schedule the tasks to mask the reconfiguration overhead.

It is also worth noting that SMASH is able to efficiently exploit available hardware resources, occupying always more than 90% of them and never violating the total area constraint.

In conclusion, the results show that the proposed methodology is effectively able to support the designer in the development of reconfigurable architectures, limiting the impact of the decisions performed by the designer about the configuration of the initial architectural template in terms of hardware modules.

VII. Conclusions and Future Work

This paper presented SMASH, a heuristic and iterative methodology for supporting the design of reconfigurable embedded systems. The methodology have been tested using a set of synthetic task graphs of different size with respect to different architectural templates. It proved to be effective, overcoming the classical limitation in the design of such systems, where the designer is faced with the problem of manually deciding the structure of the architecture.

Future works will focus on two aspects: the first one is the integration of multiple metrics to optimize multiple objectives as for instance performance and power consumption; while the second consists in the integration of a floorplanning phase to identify also the physical constraints of the resulting modules.

TABLE III: Results in terms of architectures (number of static IPs and reconfigurable regions - *IPs* and *RRs*), along with number of hardware tasks (*HW tasks*) and required reconfigurations (*#Reconf*).

	Static				*Mixed*				*Reconfigurable*			
#Tasks	*IPs*	*RRs*	*HW tasks*	*#Reconf*	*IPs*	*RRs*	*HW tasks*	*#Reconf*	*IPs*	*RRs*	*HW tasks*	*#Reconf*
12	7	0	7	0	7	0	7	0	6	0	6	0
20	20	0	20	0	18	1	20	1	17	1	19	1
31	30	0	30	0	20	4	31	7	16	7	30	7
41	30	0	30	0	18	8	40	14	12	12	40	16
52	30	0	30	0	17	9	51	25	8	17	51	26
60	30	0	30	0	15	10	53	28	10	14	51	27
70	30	0	30	0	17	9	55	28	9	16	58	33
83	30	0	30	0	15	11	80	54	6	19	81	56
90	30	0	30	0	23	3	31	5	9	12	39	18
100	30	0	30	0	16	7	46	23	3	17	53	33

Acknowledgments

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

References

[1] G. Nicolescu, I. O'Connor, and C. Piguet, *Design technology for heterogeneous embedded systems*. Springer, 2012.
[2] M. Duranton, D. Black-Schaffer, S. Yehia, and K. De Bosschere, *The HiPEAC Vision*, M. Duranton, Ed., 2011.
[3] 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.
[4] 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.
[5] 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.
[6] 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.
[7] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," in *Proceedings of CODES+ISSS '07*, 2007, pp. 9–14.
[8] Y. M. Lam, J. Coutinho, W. Luk, and P.-W. Leong, "Mapping and scheduling with task clustering for heterogeneous computing systems," in *Proceedings of FPL '08*, 2008, pp. 275–280.
[9] 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.
[10] 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.
[11] C. Bolchini, A. Miele, and C. Sandionigi, "Automated Resource-Aware Floorplanning of Reconfigurable Areas in Partially-Reconfigurable FPGA Systems," in *In Proceedings of FPL '11*, 2011, pp. 532–538.
[12] Synopsys, Inc., "Platform Architect," http://www.synopsys.com/Systems/ArchitectureDesign.
[13] "Xilinx Vivado Design Suite, available at http://www.xilinx.com."
[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] "List scheduling with and without communication delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321 – 1344, 1993.
[16] "Partial Reconfiguration Cost Calculator (PRCC), available at http://users.isc.tuc.gr/ kpapadimitriou/prcc.html ."
[17] Xilinx Inc., "Microblaze processor reference guide," 2012.
[18] Avnet Design Services ZedBoard, "http://www.zedboard.org."