Ant Colony Optimization for Mapping, Scheduling and Placing in Reconfigurable Systems

Fabrizio Ferrandi, Pier Luca Lanzi,
Christian Pilato, Donatella Sciuto
Politecnico di Milano - DEIB
Milano, Italy
{ferrandi,lanzi,pilato,sciuto}@elet.polimi.it

Antonino Tumeo
Pacific Northwest National Laboratory
Richland, WA, U.S.A.
antonino.tumeo@pnnl.gov

***Abstract*—Modern heterogeneous embedded platforms, composed of several digital signal, application specific and general purpose processors, also include reconfigurable devices supporting partial dynamic reconfiguration. These devices can change the behavior of some of their parts during execution, allowing hardware acceleration of more sections of the applications. Nevertheless, partial dynamic reconfiguration imposes severe overheads in terms of latency. For such systems, a critical part of the design phase is deciding on which processing elements (*mapping*) and when (*scheduling*) executing a task, but also how to *place* them on the reconfigurable device to guarantee the most efficient reuse of the programmable logic. In this paper we propose an algorithm based on *Ant Colony Optimization* (ACO) that simultaneously executes the scheduling, the mapping and the linear placing of tasks, hiding reconfiguration overheads through pre-fetching. Our heuristic gradually constructs solutions and then searches around the best ones, cutting out non-promising areas of the design space. We show how to consider the partial dynamic reconfiguration constraints in the scheduling, placing and mapping problems and compare our formulation to other heuristics that address the same problems. We demonstrate that our proposal is more general and robust, and finds better solutions (16.5% in average) with respect to competing solutions.**

I. Introduction

Heterogeneous Multiprocessor Systems-on-Chip (MPSoCs) are the de facto standard for modern embedded system design. They are usually composed of several general purpose, digital signal and application specific processors. To further enhance their performance, they often integrate reconfigurable devices, which allows implementing customized hardware acceleration for some sections of the target applications. The latest Field Programmable Gate Arrays (FPGAs) [1] support *Partial Dynamic Reconfiguration* (PDR). PDR allows changing portion of their configuration, while the rest of the device remains active, enabling the reuse of the device area to accelerate even more sections of an application. Nevertheless, its support imposes severe constraints and overheads, especially in terms of reconfiguration latencies and processing elements to drive the reconfiguration. Furthermore, accurate placement of the hardware components is critical in obtaining efficient utilization of the available reconfigurable area.

When developing a heterogeneous MPSoC, the designer aims at finding the best assignments for the *tasks* of the application to the available processing elements, minimizing the program execution time and, thus maximizing the performance. The designer determines when (*scheduling*) and where (*mapping*) the tasks should start their execution, depending on their resource requirements and data dependencies. However, scheduling and mapping are *NP-complete* problems [2]. PDR support with the related overheads increases the size of the exploration space, thus making exact algorithm unpractical. Consequently, researchers employed heuristics based on Genetic Algorithms [3], Simulated Annealing, Tabu Search [4], [5], Kernighan-Lin [6] for obtaining sub-optimal solutions to these problems in reasonable times. *Ant Colony Optimization* (ACO) [7], [8], [9] is another heuristic approach that has been recently applied to these problems with good results. However, the majority of the proposed formulations try to separately solve the various problems. If they simultaneously address the different problems, instead, they either are limited to few tasks and few implementations points [10] or do not consider PDR and placing [9].

This paper proposes an ACO-based heuristic algorithm that overcomes most of these limitations. In particular:

1) it simultaneously performs scheduling, mapping and placing of a task graph on a complex and heterogeneous MPSoC with reconfigurable devices that support PDR;
2) it can find sub-optimal solutions independently from the fact that a task can execute on different resources or requires the availability of multiple resources for execution;
3) it accounts for the reconfiguration constraints and overheads, such as the reconfiguration latency, the presence of a reconfiguration port and the execution of the reconfiguration task on a general purpose processor, potentially exploiting configuration pre-fetch.

The paper proceeds as follows. Section II introduces the problem addressed in this work and ACO. Section III details the formulation proposed in this paper. The formulation is then compared with existing work in Section IV. Section V presents the experimental evaluation, comparing our algorithm to previous heuristics dealing with the same problems. Finally, Section VI concludes the paper.

978-1-4673-6383-9/13/$31.00 ©2013 IEEE

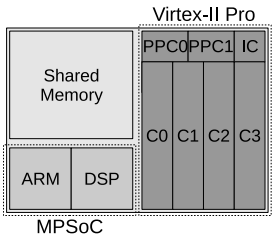

Fig. 1: The model of the target architecture

II. Background

This section introduces the problem addressed in this work through a motivating example and the ACO heuristic used to perform the design space exploration.

A. Definitions and Motivating Example

This work targets a MPSoC composed of multiple general purpose processors (not necessarily with the same instruction sets), digital signal processors and FPGA devices that support one-dimensional reconfiguration. An example of such a MPSoC is shown in Figure 1.

FPGAs can be described as a matrix of Configurable Logic Blocks (CLBs), interconnected through switches, that perform the programmed functions. PDR is one of the most powerful mechanisms to configure modern FPGAs: it allows reconfiguring only a portion of the system without interrupting the operations in the other parts. FPGAs with such features are, for example, the devices of the Xilinx Virtex family. The Virtex-II Pro and the Virtex-4 devices, respectively have a basic reconfiguration block of an entire column of CLBs or of a sub-matrix of $16x1$ CLBs. Newer solutions, such as the Virtex 5, 6 and 7, support the same features, and only differ for the reconfiguration granularity. Reconfiguration time is proportional to the number of CLBs reconfigured. PDR is performed by accessing a specific reconfiguration port called Internal Configuration Access Port (ICAP). Some devices even integrate hard-core processors that can drive the self-reconfiguration, and heterogeneous elements like multipliers and memories. Some versions of these FPGAs even host multiple (up to 2) ICAPs and several (up to 4) hard-core processors, such as the Power PC 405. To reconfigure such devices, a processing element fetches the new configuration for some of the CLBs from an external memory, and then sends it to the ICAP. After the reconfiguration time, the reconfigured hardware can then receive data and can start operating.

Figure 2 shows the Direct Acyclic Graph (DAG) that represents the data dependencies among the tasks. Suppose that some of these tasks can run, with different performances, on the various processing elements available on the target platform. There are specific software implementations of the task for the ARM, the DSP, the PowerPC, and, possibly, multiple hardware implementations for the FPGA with different area/performance trade-offs. The objective is obtaining the minimum execution time (maximum performance) for this application on the target platform. There are multiple problems to solve. The designer must decide on which processing elements

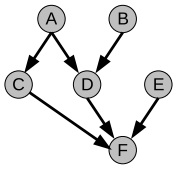

Fig. 2: Sample task graph

TABLE I: Distribution of the execution times and area occupation on the different resources of the architectures for the example task graph. 4 columns are available on the FPGA.

Task	ARM	DSP	PPC	HW T.	HW A.
A	8	6	2	1	1
B	5	6	10	2	2
C	4	4	6	1	2
D	8	7	2	3	1
E	10	8	3	1	1
F	2	3	7	1	2

he should map the tasks. The designer should also decide in which order the system should start the tasks, considering their dependencies. Suppose that, for performance reasons, the designer mapped two potentially parallel tasks (A and B) on the same resource. Scheduling A before B makes possible launching C without waiting for B completion. Then, consider the possibility that some tasks have a promising hardware implementation, and the designer would like to run them on the FPGA. If the FPGA does not support PDR, there is a constraint on the available area. She or he can allocate only a few of the tasks on the FPGA, and a careful selection is required to obtain the best application execution time. However, if the FPGA supports partial dynamic reconfiguration, more possibilities for hardware implementation can be exploited, but new constraints on the placing and on the reconfiguration overheads appear, making the decision process even more complex. Consider the case of columnar (linear) placement and suppose that tasks A, B, C and E are selected to run on the FPGA. Resource requirements and performance of the tasks are reported in Table II. Figure 3 shows some possible placings for the example. In the figure, columns represent the physical columns available on the device, and rows represent the passing of time. As shown in *i)*, if A, B and E are placed in that order, C can only start at time 4, since only at that time 2 consecutive columns become available. If A and E are placed before B, instead, there are 2 consecutive columns for task C already available at time 2, as shown in ii. In *ii)*, however, reconfiguration overheads are not considered. On a FPGA supporting PDR, the situation is more similar to the one depicted in *iii)*. After A and E end, reconfiguration of columns 0 and 1 for C (rC) starts. To perform reconfiguration, a processor is needed to fetch the configuration from memory, and to send it to the ICAP. Thus, both the processor and the ICAP are occupied for the reconfiguration time. Furthermore, the columns used by task C are also blocked during reconfiguration time, because they are being loaded with the new function. They will be free again only at the end of task C.

This example shows how the spawning of a hardware task on the FPGA can be considered as a combination of a recon-

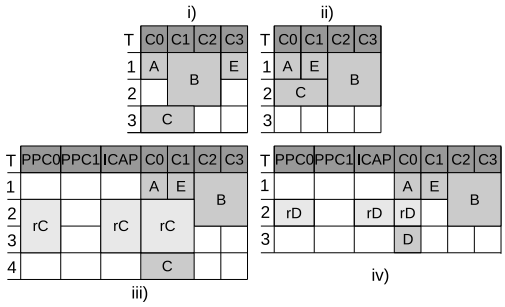

Fig. 3: Resources reuse on FPGA supporting columnar reconfiguration. A) shows a basic placing, B) shows a more effective placing, C) represents the real case for B) when partial dynamic reconfiguration is involved.

figuration and an execution component. The reconfiguration component actually represents the reconfiguration overhead and, to be performed, requires a *set of resources*, composed of a processing unit to drive the reconfiguration, the configuration port, and the programmable blocks. The execution part must start only when the data dependencies of the tasks have been satisfied (i.e. all the predecessors have ended and the data required by the tasks are ready). In some cases, instead, it is possible to anticipate the reconfiguration, if the required set of resources (processor, configuration port and columns) is available before the dependencies have been satisfied. So, it is not necessary to wait for the termination of the predecessors to start the reconfiguration of a task. The reconfiguration component leaves the reprogrammed columns unavailable until the termination of the execution part. The only constraint is that no other reconfigurations may happen on those columns until the related execution component terminates. However, the execution component does not necessarily need to start right after reconfiguration. This situation is shown in Figure 3, *iv)*: reconfiguration of D (rD) can start before the end of B, and D can start its execution as soon as B is terminated. In this way, the reconfiguration overheads for D are completely hidden. This technique is called configuration pre-fetch [11], and it is possible only with adequate placing and scheduling.

Efficient solutions can be found only by analyzing the scheduling, the mapping and the placing of the tasks at the same time and effective methods to search in the design space are definitively required. For these reasons, this work proposes an efficient heuristic, based on ACO, to solve the combined scheduling, mapping and placing problem for heterogeneous MPSoCs augmented with FPGAs that support PDR. Our algorithm performs hardware-software partitioning on complex architectural models that feature many heterogeneous processing elements, considers the reconfiguration and placing constraints, and returns feasible scheduling, mapping and placing of the initial task graph.

B. Ant Colony Optimization

Ant Colony Optimization (ACO) is a heuristic search methodology originally introduced by Dorigo *et al.* [12] with the Ant System (AS). ACO mimics the cooperative behavior of ants when searching for food. Ants start from their nest going in random directions, depositing a trail of *pheromone* that motivates other ants to follow the same path. The ants moving on the shortest paths will reach the food and come back faster than the others, proportionally depositing more pheromones and reinforcing the trails. As time passes, the oldest trails evaporate, and, at some point, only the shortest path will remain with a strong reinforcement. ACO takes inspiration from this behavior. Several agents are launched to explore the search space, taking a sequence of decisions to find a solution. At each step, an agent only takes admissible moves. It then reinforces each decision proportionally to the quality of the resulting solution. The collaboration of multiple agents allows to concentrate the exploration only on the promising zones of the search space (global heuristic). For each choice, the agents also exploit local heuristics that are directly related to the problem.

AS has initially been applied to the Traveling Salesman Problem (TSP). In the TSP, the cost of each tour is the sum of the costs of each arc chosen to go from the first vertex to the last one. On the TSP, AS works as follows:

1. Initially associate each arc with a pheromone trail τ_{ij}.

2. Put *m* ants on an initial vertex.

3. Each ant constructs its own tour, executing a probability choice at each step from the set of allowed c and memorizing the visited cities. The probability of going from vertex i to j is calculated as:

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha} * [\eta_{ij}]^{\beta}}{\sum_{l \in N_i} [\tau_{il}]^{\alpha} * [\eta_{il}]^{\beta}}$$

where N_i is the set of admissible choices for the ant at vertex i, η is the local heuristic that influences the choice of the ant on the next arc to explore starting from the current node, τ is the pheromone trail, while α and β controls the weight of the local heuristic and the global heuristic (pheromones). The TSP formulation uses a very simple "greedy" local heuristic (the inverse of the cost of an arc), which guides the ants to the most promising solutions since the beginning of the search.

4. Evaluation of the quality of the result.

5. Update of the the pheromones, firstly evaporating the trails on all the arcs and then incrementing it of a factor proportional to the quality of the result found.

6. If goal conditions are not met, go to step 2.

The pheromones update formula is the following:

$$\tau_{ij} = (1 - \rho) * \tau_{ij} + \sum_{l=l}^{m} \Delta\tau_{ij}^{(l)}$$

where $0 < \rho < 1$ is the evaporation rate and the deltas are calculated as$\Delta\tau_{ij}^{(l)} = Q/L$, if the arc ij was in the solution, 0 if not, with Q as pheromone delivery rate and L representing the cost of the result. Evaporation removes, after each iteration, some pheromones on all the arcs. This allows forgetting arcs explored and never revisited, avoiding early convergence to local minima. Termination criteria of the search can be a maximum number of generations or convergence to a sub-optimum.

III. Proposed Algorithm

In contrast with many formulations for mapping and scheduling problems, our algorithm starts from the concept that it is actually performing a heuristic scheduling. In many scheduling approaches, heuristics simply optimize the priority function of a classic list formulation. In other cases, list scheduling is used after the heuristic search to evaluate the quality of the mapping chosen for the tasks. List scheduling exploits a heuristic priority function also in these cases. Our ACO formulation, instead, focuses on the concept that at every decision point an ant decides which task to schedule and where to map it. Thus, every ant constructs step by step the mapped and scheduled task graph, choosing one node after the other. The basic idea comes from [13], which applies ACO to the Resource Constrained Scheduling Problem (RCSP). Objective of such work is to find the best schedule (shortest execution time) for a series of dependent jobs (described through a DAG) that required pre-determined sets of resources. Each ant constructs a complete schedule in N steps, where N is the number of jobs, following a Serial Generation Scheme (SGS). At (SGS). At each step, the ant selects a new job from a candidate list. The candidate lists includes all the jobs that have satisfied dependence constraints, and for which all the required resources are available at the current timeframe. When a job is selected, the availability of the related resources is updated to the current scheduling time plus the execution time of the selected job. With appropriate choices, SGS can always reach the optimal solution for the RCSP [14]. Following the ACO approach, each ant generates its own scheduling. The results (overall execution times) are then evaluated, and the pheromone matrix (which stores the feedback about the choice of a node at a scheduling step, thus has size NxN) is updated following the standard update policies. In subsequent iterations, the ants will converge to the shortest schedules.

We adapted this formulation to perform scheduling, mapping and partial dynamic configuration. In doing so, we introduced support for multi-stage decisions (that reduce the exploration space) and support for unknown job duration (reconfiguration tasks locks the columns until the execution tasks are scheduled). We present our algorithm with an example.

A. Algorithm presentation

Consider the sample task graph of Figure 2 and the target architecture of Figure 1. Table II reports the execution time the tasks on the processing elements of the architecture, the performance of their hardware implementation and the area (columns) occupied on the FPGA. The numbers are simplified for the example, but maintains realistic ratios. Figure 4 shows the steps performed by an ant to schedule, map and place the sample task graph on the target architecture with support for self-reconfiguration.

In our formulation, an ant starts selecting tasks from those that have not dependencies. However, as previously discussed, tasks that can execute on the FPGA are decoupled in reconfiguration and execution tasks. Reconfiguration tasks are special tasks, that do not depend from any other tasks. However, the related execution tasks directly depends on them. They also require the availability of a set of resources: a processor, the ICAP and the columns of the FPGA. To simplify the example, we use a reconfiguration time of 1 time unit for each column (space unit) occupied by a task. Thus, at step *(i)* of our example, all the reconfiguration tasks are schedulable, because at the beginning all the required units are free. Task A, B and E are also schedulable, because they are not dependent on any other task. Nevertheless, they conceptually cannot run on the FPGA, because a reconfiguration is required. This is only a requirement for the formalization of the algorithm, because our approach also consider the possibility to start with a predetermined configuration for the FPGA.

TABLE II: Distribution of the execution times and area occupation on the different resources of the architectures for the example task graph. 4 columns are available on the FPGA.

Task	ARM	DSP	PPC	HW T.	HW A.
A	8	6	2	1	1
B	5	6	10	2	2
C	4	4	6	1	2
D	8	7	2	3	1
E	10	8	3	1	1
F	2	3	7	1	2

To deal at the same time with the mapping, the scheduling and the placing problem, the ant would have to choose among all the possible placing of each reconfiguration task, and among all the possible implementations for all the other tasks, with the exception of the FPGA. Since this would generate a very large set of admissible moves, we decoupled the decision process in two stages. In the first stage, the ant only chooses which task to schedule among the reconfiguration tasks and the other available tasks (not running on the FPGA). In the second stage, it either decides the placing (if a reconfiguration task is selected) or the mapping (if any other task is selected). In subsequent steps, if a reconfiguration task was selected, the related execution task will still appear in the list of admissible selections. If the related execution task is selected in the first stage, in the second stage it will be simply set as executed on the previously decided FPGA columns.

Suppose that the ant selects reconfiguration of A (rA) as the first task to schedule. The second stage of the decision process determines on which columns the task should be placed and which processors can drive the reconfiguration. Supposing that only the processor in the FPGA can drive the reconfiguration, we can place A on column 0, 1, 2, or 3 using PPC0 (first column of "options" in the figure) or PPC1 (second column of "options"). Note that there is only one ICAP, so it is implicitly used for any of the choices. However, at this point, the FPGA has not been used yet, and reconfiguration is not really needed (the system can be set up to start with an initial configuration). So, the ant does not occupy all the processors, but simply set the selected column (C0) as blocked. At step *ii)*, since only one column is currently unavailable, and the hardware implementations of all the tasks occupy 2 or fewer columns, all the remaining reconfiguration tasks can be scheduled, again with the execution for tasks A, B and E. Anyhow, the ant selects the execution task for A. Consequently, in the second

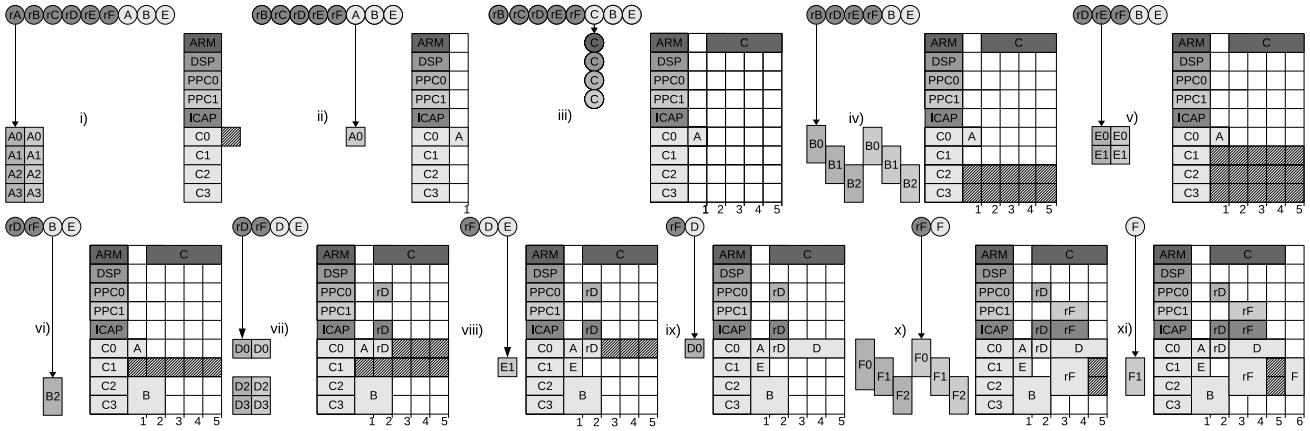

Fig. 4: An iteration of our ACO algorithm with support for self-reconfigurable devices

stage of the decision process, it is placed on the column that was already blocked. C0 will be now available again for other tasks starting from time 1. The execution of Task A satisfies the dependencies of Task C, which thus becomes a candidate for execution. The ant, at step *iii)*, chooses it. Since no reconfiguration of C has been scheduled before, the second stage of the decision only involves the software implementations. C is scheduled on the ARM at time 1, and automatically rC is removed from the set of possible scheduling candidates. At step *iv)* the reconfiguration task for B is chosen. No columns are currently blocked by other reconfigurations, so the ant has all the options to place B, which has size 2: columns 0-1, 1-2, or 2-3. The ant can also choose if reconfiguration can be driven by PPC0 or PPC1. Driven by a local heuristic, which motivates the ant to choose the column which allows to start the task as soon as possible, and by past ants through the global pheromones matrix, B is placed on column 2-3. Again, no reconfiguration time is considered, since those column where not used before.

At step *v)*, column 2 and 3 are blocked. However, it is possible to schedule reconfiguration of tasks of size 1 (D and E) and of size 2 (F, but only starting from the end of A). The ant chooses to schedule rE, and the second stage can place it on column 0 (but with reconfiguration), or on column 1 (without actual reconfiguration) on PPC0 or PPC1. The heuristic chooses column 1, and sets it as used. At step *vi)* the execution task for B is selected, and so the time at which columns 2 and 3 becomes again available again is set to 2. At step *vii)*, the execution task for D becomes available, but the ant selects its reconfiguration task, rD. Feasible placings at this round are column 0, 2 and 3. Again, the ant can also choose which processor will drive the reconfiguration. This time, however, all the feasible columns were already occupied by other tasks, so reconfiguration has an actual duration of 1 unit of time. The ant chooses to map the reconfiguration on column 0. This is the most convenient column (1 is not available, while in 2 and 3 the reconfiguration would start at time 2). The reconfiguration is then scheduled on the set of resources composed by the PPC0 (driving the reconfiguration), the ICAP (reconfiguration port) and the target column at time 1. Reconfiguration of D has not any data dependencies from B (only the execution has) so it can be scheduled as soon as the three required resources are available. After reconfiguration, column 0 remains locked until execution of D is scheduled. Execution of D is scheduled after the end of the execution of B, and the net effect is that the reconfiguration has been pre-fetched without extending the schedule. At step *viii)* and *ix)* the execution tasks of E and D are scheduled. At step *x)*, execution of F becomes available, but rF is instead chosen. Since no columns are blocked by another reconfiguration task, the ant has all the possible placings. However, starting to place the task in column 0, would mean to delay it until time 5. The heuristic then opts to reconfigure columns 1 and 2. The task has size 2, so reconfiguration lasts on the driving processor, the ICAP and the 2 columns, for 2 units of time. At step *xi)* the only remaining task to schedule is the execution of F. The ant chooses it, but the scheduler sees that it has to be started at the end of D for data dependencies. Thus, at time slot 5 the two columns are already loaded with the new task, but they are not used for execution since dependencies are not yet satisfied. In this case, reconfiguration pre-fetch also results in an unused time slot.

The example also identifies the interesting aspects of the algorithm: it performs mapping, scheduling and placing at the same time, constructing, step by step, only complete and feasible solutions and exploring around them. The scheduling mechanism supports resources for which availability time is not known, operations (tasks) which can run only on specific resources (i.e. only some processors can drive reconfiguration) and operations which requires a set of resources to run (the reconfiguration tasks need a set of resources to run). It naturally supports the limitation due to a single reconfiguration port and exploits, where possible, reconfiguration pre-fetch. We presented the algorithm considering a target architecture where data are exchanged through a single shared memory, and thus, supposing low interference, communication costs can be directly included in task execution times adding them an overhead. However, we underline that the proposed algorithm can be extended to support communication tasks. This may be useful, for example, for architectures where components

can communicate through different communication channels. The extension is trivial, as communication tasks directly correspond to the arcs connecting the vertices (execution tasks) in the task graph. The communication tasks can can be mapped only on some specific communication resources, depending on where the source execution task has been mapped (and thus on the available communication channels of the chosen resource), and must be scheduled before the dependent execution task. We also focused the example on 1D reconfiguration, but 2D reconfiguration is also easily supported. However, the search space increases depending on all the possible allocations of a hardware implementation.

B. Algorithm details

The multistage decision process determines certain trade-offs with respect to a single stage decision process where the and at each step has all the possible permutations of tasks with mapping for software implementations or tasks for placing with hardware implementations. First, it allows considering reconfiguration with pre-fetch and placing, without generating too many candidates for all the admissible solutions (all tasks with satisfied dependencies on all the processors, plus all the reconfiguration tasks with all the possible placings). Second, it reduces the dimensions of the pheromone data structures. With a single decision process, the pheromone matrix would have size $2Nx2Nx(P + C)$. In fact, there potentially are reconfiguration and execution tasks $(2N)$ for each one of the original tasks (N), that can be selected in any of the up to $2N$ scheduling steps, P resources (processors without the FPGA) and C FPGA columns. By separating the decision process, we require a data structure of $2Nx2N$ for the "scheduling pheromones" (τ^s) and a data structure of $2Nx(P+C)$ for the "mapping and placing pheromones" $(tau^m$, with $M = P+C)$. These can be compared to [13], where only scheduling is performed (mappings are predetermined for each job), with a pheromone data structure of NxN. Finally, using a multi-stage decision process allows makes possible to use local heuristics specific to the each problem, while maintaining a good correlation among scheduling, mapping and placings, because the ant is still exploring all the dimensions in a single step and the pheromone feedback for subsequent iterations is related to all the choices.

For the decision process, probabilities of selecting a node for scheduling, and subsequently for mapping and placing are as follow. The probability of scheduling task t at step i is:

$$p^s_{it} = \frac{[\tau^s_{it}]^{\alpha^s} * [\eta^s_{it}]^{\beta^s}}{\sum_{l \in N_i} [\tau^s_{il}]^{\alpha^s} * [\eta^s_{il}]^{\beta^s}}$$

Schedulable tasks t include both reconfiguration and execution tasks. The local heuristic η^s is the mobility of the task. If the ant schedules an execution tasks for which a reconfiguration tasks was not selected, the probability to map task t to resource m (only resources P are selected from M) is:

$$p^m_{tm} = \frac{[\tau^m_{tm}]^{\alpha^m} * [\eta^m_{tm}]^{\beta^m}}{\sum_{l \in N_t} [\tau^m_{tl}]^{\alpha^m} * [\eta^m_{tl}]^{\beta^m}}$$

The local heuristic η is Earliest Finish Time (EFT) of the task on the selected resource. If, instead, the ant selects a reconfiguration task t, the probability to map it on the set of resources that includes the processor driving the reconfiguration, the ICAP and the set of columns (potentially all over M, because any processor could drive the reconfiguration) is:

$$p^m_{tm} = \frac{[\tau^m_{tm}]^{\alpha^m} * [\eta(p)^m_{tm}]^{\beta^m}}{\sum_{l \in N_t} [\tau^m_{tl}]^{\alpha^m} * [\eta(p)^m_{tl}]^{\beta^m}}$$

Where $\eta(p)$ is calculated as an average among the pheromones deposited for selecting the reconfiguration of the task for each one of the required resources.

Pheromones update is performed by saving the "trace" of the decision process of the ants (i.e., for each step we save the all the selections in a ordered list), so we know the step in which a task has been scheduled and the resource on which the task has been mapped. At the end of every iteration, the algorithm reinforce the pheromone trails in the two data structures for the best ant of the current iteration and for the overall best found following the elitist policy. To reduce the possibility of getting stuck in local minima due to continuous reinforcement, the overall best solution also can, with low probability, be replaced by the current best.

IV. Comparison with Related Work

Exact Integer Linear Programming (ILP) formulations for scheduling [15] and hw/sw partitioning [16] have been proposed, but they are not applicable for large instances of the problems. Generally, heuristics method are preferred to obtain sub-optimal results in acceptable times. A common approach for the Resource Constrained Scheduling problem (RCSP) is the list based algorithm [17], which uses a priority list to determine the order in which operations should be scheduled. The priority list can be obtained with several methods, including optimization heuristics like Simulated Annealing (SA), Tabu Search (TS) [18] and Genetic Algorithms (GAs) [3]. GAs [19], TS and SA [4], [5] have also been used to solve hw/sw partitioning and mapping problems. The Kernighan-Lin-Fiduccia-Mattheyses (KLFM) heuristic [20], which tries to find the best solution performing local moves, has been successfully adopted [6] in many formulations. These works, however, assume that the hardware is static, i.e. the programmable components cannot be reconfigured. ACO has been recently applied separately to both scheduling [7], [13] and hw/sw partitioning [21] for multiprocessor systems with static programmable logic. Our formulation, however, differs from these works for several aspects. We consider scheduling and mapping simultaneously, we deal with placing constraints on the reconfigurable components and we address PDR and its overheads. Recently, proposals to solve several of these problems together have been made. [22] and [23] deal with scheduling and placing at the same time for devices supporting PDR. However they do not consider some of the constraints that PDR imposes, such as the presence of a single configuration port, the requirement of a processor to drive the reconfiguration, and the possibility to perform

reconfiguration pre-fetch. A solution which may resemble ours is the one proposed by Banerjee [10] *et al.*. This work introduces an ILP formulation that considers scheduling and linear placing on devices with partial dynamic reconfiguration, along with a KLFM-based heuristic. The limitations due to the presence of a single reconfiguration port and the requirement of a processor are addressed, and reconfiguration pre-fetch is considered. Also, tasks can have multiple hardware implementations. Compared to this work, our solution is more general. We deal with a multiprocessor platform, and beside supporting multiple implementation points for hardware tasks, we also address multiple software versions. Furthermore, we perform considerably more exploration on the possible solutions for the problem. KLFM methodology [10] performs a scheduling for each possible implementation point of the tasks and highly depends on the initial (random) partitioning to obtain good results. Scheduling is performed with a list based algorithm, and the priority list generation adopts a heuristic that considers also placing constraints for hardware tasks. The main difference is that our ACO heuristic is wrapped by the scheduler, and thus each step is actually a scheduling step. In Banerjee's formulation, instead, the scheduler is wrapped inside the KLFM heuristic, and it is used to evaluate the benefit of moving a task to another implementation point. However, the placing is addressed only by the heuristic of the scheduler, which is fixed, and can greatly reduce the search space for the algorithm. Even if, in our case, the search space is bigger, thanks to the combined local and global heuristics of ACO we can limit our exploration only in the promising zones, without necessarily evaluating all the possible moves. In the following section, we use this heuristic as a term of comparison for the simplified case of a single processor and a self-reconfigurable device. In [9] an ACO approach that explores, together, mapping and scheduling of tasks and communications is presented. The approach follows some of our solutions (multistage decision approach, elitist policy) but it does not account for PDR. We use this approach as a term of comparison for both the simplified case and for the case of a full MPSoC augmented with a FPGA supporting PDR.

V. Experimental Evaluation

To evaluate our heuristic, we generated several task graphs of varying dimensions using TGFF [24]. In particular, we compared our ACO formulation with support for PDR to [9] and [10], targeting a simplified architecture. This architecture, similar to the one proposed in [10], supports a single general purpose processor, a single reconfiguration port, and 20 columns of reconfigurable logic blocks. Task annotations have been generated considering that a software implementation can be around 3 to 5 times slower than a hardware implementation. Thus, hardware tasks take 1600 ± 1500 cycles, while software tasks require 8600 ± 1500 cycles. Reconfiguration time is proportional to the occupation of the tasks. For our benchmarks, a task occupies from 3 to 7 columns. We supposed that each column takes 300 cycles to be reconfigured. We also modeled communication costs, considering a 100 cycles overhead when there is a data transfer from software to hardware tasks and vice-versa. Parameters for our ACO algorithm were, respectively, 0.1 and 0.02 for scheduling and mapping/placing evaporation rate. We used a number of ants proportional to the number of nodes, and launched as many iterations as required to make comparable the execution times, if our algorithm did not reach convergence to a result before. The algorithm returns the schedule with the lowest execution time (number of cycles) found.

TABLE III: Comparison among [9] (ACO without support for self reconfiguration), [10] (KLFM based heuristic) and this work (ACO with support for self reconfiguration) on a simplified target architecture.

#Tasks	[9]	[10]	This work	Gain wr [9]	Gain wr [10]
10	28,081	15,629	9,997	64.4 %	36.0 %
20	84,182	27,325	22,113	73.7 %	19.1 %
30	113,430	43,922	37,109	67.3 %	15.5 %
40	126,386	62,366	58,149	54.0 %	6.8%
50	167,224	74,469	70,769	57.7 %	4.9%
Total				63.4 %	16.5 %

TABLE IV: Comparison between [9] and this work on a MPSoC augmented with self-reconfigurable logic

#Tasks	[9]	This work	Gain
10	19,260	11,163	42.0 %
20	37,392	21,595	42.3 %
30	63,048	42,001	33.4 %
40	95,405	63,225	33.7 %
50	108,022	81,938	24.1 %
60	137,414	112,794	17.9 %
70	161,290	128,343	20.4 %
80	185,702	159,391	14.2 %
90	225,382	188,648	16.3 %
100	232,561	189,836	18.4 %
200	503,546	446,127	11.4 %
250	670,282	559,554	16.5 %
500	1,296,583	1,095,948	15.5 %
Total			23.5 %

Table III shows the results for task graphs going from 10 to 50 tasks. Our ACO formulation performs considerably better, in particular with small task graphs. The reason is that the algorithm in [10] mainly explores the assignment of tasks to hardware and software, while, for placing, it simply adopts a greedy heuristic when determining the priority of a task in the list based scheduler. So, even if it considers reconfiguration pre-fetch, sometimes it cannot find some interesting placing solutions. Our algorithm, instead, can also explore in that direction, eventually finding placements with less fragmentation that allow a better reuse of the reconfigurable logic. When the number of tasks grows, however, the search space becomes larger and the benefits of our method decrease. We can then allow the ACO method to run for longer times and find better results, or raise the evaporation rate to limit the global search and focus more on local exploration. The same table also shows the comparison, on the same architecture, to [9], which does not support self-reconfigurable devices. On this experimental setup, with a single processor and no possibility to reuse the reconfigurable logic, our approach can find schedules with 64.5% in average better execution times. A key issue is that, with the proposed modeling of the problem, multiple reconfigurations cannot go in parallel, but are *sequentialized* by the presence of a single reconfiguration

port (ICAP). Furthermore, for the reconfiguration time, the general purpose processor is used to drive the reconfiguration itself. Maximum gain is obtained for the 20 nodes task graph (73.7%), while the minimum speed up (54%) is verified with the 40 nodes graph. The variability of the results depends on the size of the problem, since a bigger task graph gives to our ACO formulation more opportunities to explore for configuration pre-fetch and placing of the tasks, but also from the fact that the heuristic may find less opportunities to pre-fetch the reconfiguration with larger instances.

This situation is better detailed in Table IV, which compares the results of [9] to our ACO formulation on the complete target platform addressed by this work, composed of a DSP, an ARM processor, two PPCs, an ICAP and 20 columns of reconfigurable logic. Task graphs have again been generated with TGFF. Since the DSP can be very fast on some tasks but very slow for others, we generated performance annotations of 7000 ± 3500 cycles for it. For the ARM processor, we supposed instead similar performance to the PPC, with a slightly lower average (8000) but the same variance (± 1500). We also modeled communication costs from the couple ARM-DSP to the PPCs and to the hardware tasks (100 cycles overhead). Our ACO formulation obtains schedules with 23.5% better execution time in average for all the 13 considered task graphs. Again, on smaller task graphs it is easier for our heuristic to find better placements to exploit reconfiguration and configuration pre-fetch, and thus to extract more parallelism. With larger task graph instances (over 50 nodes), the exploration space for our ACO formulation grows, but it constantly allows obtaining schedules from 10% to 20% better, depending on how pre-fetch can be scheduled. We should underline that also in this architecture only a single reconfiguration at a time can be performed, because there is a single ICAP, but the reconfiguration can be driven by any one of the 2 PPCs.

VI. Conclusions

In this paper, we described an ACO heuristic for scheduling, mapping and placing applications on heterogeneous MPSoCs augmented with FPGAs that support PDR. Our heuristic takes into consideration the overheads imposed by PDR in terms of latency, fragmentation of the hardware tasks, limited number of reconfiguration ports and the requirement of a processing element to drive the reconfiguration. We compared our ACO formulation to previous heuristics and obtained solutions 16.5% better in average on a target platform with a single processor. We then applied our methodology to a MPSoC augmented with a FPGA and compared it to an ACO-based heuristic without support for PDR. We showed that our algorithm is able to provide better results by exploiting PDR while hiding its overheads.

Acknowledgments

This work was partially funded by the European Commission in the context of the FP7 FASTER project (#287804). We also thank the Center for Adaptive Supercomputing Software - MultiThreaded (CASS-MT) for the support.

References

[1] Xilinx Virtex FPGA User Guides. Available at http://www.xilinx.com.

[2] David Bernstein, Michael Rodeh, and Izidor Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.*, 38(9):1308–1313, 1989.

[3] Martin Grajcar. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *DAC '99: the 36th ACM/IEEE conference on Design Automation*, pages 280–285, 1999.

[4] T. Wiangtong, P.Y.K. Cheung, and W. Luk. Comparing three heuristic search methods for functional partitioning in hardware–software codesign. *Design Automation for Embedded Systems*, 6(4):425–449, 2002.

[5] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2:5–32, 1997.

[6] F. Vahid and T.D. Le. Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning. *Design Automation for Embedded Systems*, 2(2):237–261, March 1997.

[7] G. Wang, W. Gong, B. DeRenzi, and R. Kastner. Ant colony optimizations for resource- and timing-constrained operation scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1010–1029, June 2007.

[8] Po-Chun Chang, I-Wei Wu, Jyh-Jiun Shann, and Chung-Ping Chung. ETAHM: An energy-aware task allocation algorithm for heterogeneous multiprocessor. In *Proceedings of DAC '08*, pages 776 –779, june 2008.

[9] 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 Computer-Aided Design of Integrated Circuits and Systems*, 29(6):911–924, june 2010.

[10] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt. Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration. *IEEE Trans. on VLSI Systems*, 14(11):1189–1202, 2006.

[11] S. Hauck. Configuration pre-fetch for single context reconfigurable processors. In *FPGA*, 1998.

[12] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[13] D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. *IEEE Transactions on Evolutionary Computation*, 6(4):333–346, August 2002.

[14] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 127(2):394–407, 2000.

[15] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI '00: ACM SIGPLAN Conference on Programming Language, Design and Implementation*, pages 121–133, 2000.

[16] R. Niemann and P. Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2(2):125–163, March 1997.

[17] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[18] S. J. Beaty. Genetic algorithms versus tabu search for instruction scheduling. In *International Conference on Neural Network and Genetic Algorithms*, pages 496–501, 1993.

[19] J. I. Hidalgo and J. Lanchares. Functional partitioning for hardware-software codesign using genetic algorithms. In *23rd Euromicro Conference*, pages 631–638, 1997.

[20] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system tech. journal*, 49(1):291–307, 1970.

[21] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Application partitioning on programmable platforms using the ant colony optimization. *Journal of Embedded Computing*, 1(12):1–18, 2005.

[22] S. Fekete, E. Kohler, and J. Teich. Optimal FPGA module placement with temporal precedence constraints. In *DATE '01: Design, Automation and Test in Europe*, pages 658–665, 2001.

[23] Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. Temporal floorplanning using the t-tree formulation. *ICCAD '04: IEEE/ACM Int'l Conference on Computer Aided Design*, pages 300–305, 2004.

[24] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES/CASHE '98: the 6th International Workshop on Hardware/Software Codesign*, pages 97–101, Seattle, WA, 1998.