Considering reconfiguration overhead in scheduling of dependent tasks on 2D Reconfigurable FPGA

Quang-Hai Khuat, Daniel Chillet
University of Rennes 1, IRISA/INRIA
Lannion, France
{Quang-Hai.Khuat, Daniel.Chillet}@irisa.fr

Michael Hübner
Ruhr-University Bochum, ESIT
Bochum, Germany
Michael.Huebner@ruhr-uni-bochum.de

***Abstract*— Configuration prefetching is known as an effective technique for hiding the reconfiguration delay of hardware accelerators in Partial Region FPGA. In prefetching, a hardware task can be loaded as soon as possible even if it cannot execute immediately after its reconfiguration due to the involvement of dependencies with other tasks. But due to the access in advance, the configuration delay is hidden. This method can be compared with a software prefetching in the processor domain. However, in the context of reconfigurable architecture, the difficulties come from the dependencies of prefetching with task scheduling and placement aspect. In this paper, we introduce an run-time spatio-temporal scheduling heuristic for dependent tasks executed on 2D heterogeneous FPGA. The objective is to reduce the reconfiguration delay of tasks, thus minimize the total execution time of an application. To achieve it, our proposed heuristic tries to prefetch tasks as early as possible while considering two factors: the priority of new tasks to be loaded and the placement decision to avoid conflicts among tasks. The experiments show that our heuristic reduces significantly the overall execution time by 22% compared to a non-prefetching method and approximately 5% compared to other prefetching methods.**

I. Introduction

One of the main research interests of FPGA is dynamic and partial reconfiguration which allows a system to change fraction of its resources at runtime without affecting the rest of the system. This feature obviously provides a higher flexibility and more powerful computing per area to deal with the dynamism of current multimedia applications requiring a high performance. Nevertheless, Partial Reconfiguration (PR) does not come for free as it may create a costly reconfiguration overhead, therefore an increase of the overall latency of the application. For instance, loading (or reconfiguring) one tenth of a Virtex XC2V6000 requires at least 4 ms or loading a JPEG decoder task which occupies 30% of this Virtex, requires 12 ms with the reconfiguration circuitry running at maximum speed [1]. Another example is : reconfiguring a device with a partial bitstream of 1.1 Mbytes on a Virtex II-Pro could take 0.92 ms [2] if the Internal Configuration Access Port (ICAP) is fed at the over-clock frequency of 300 Mhz. Thus, developing techniques to minimize the reconfiguration time overhead, making the use of PR more effective is crucial.

Among several techniques, configuration prefetching is known as an effective technique for hiding the reconfiguration delay of hardware accelerators (tasks) in Partial Region FPGA. The principle of prefetching is to load a task as soon as possible whenever the configuration controllers are available and positions are available for the task on the FPGA. Even if the task cannot execute immediately after its reconfiguration due the involvement of dependencies with other tasks, the fact of hiding the reconfiguration phase by loading the task during the execution of other tasks reduces significantly the reconfiguration overhead.

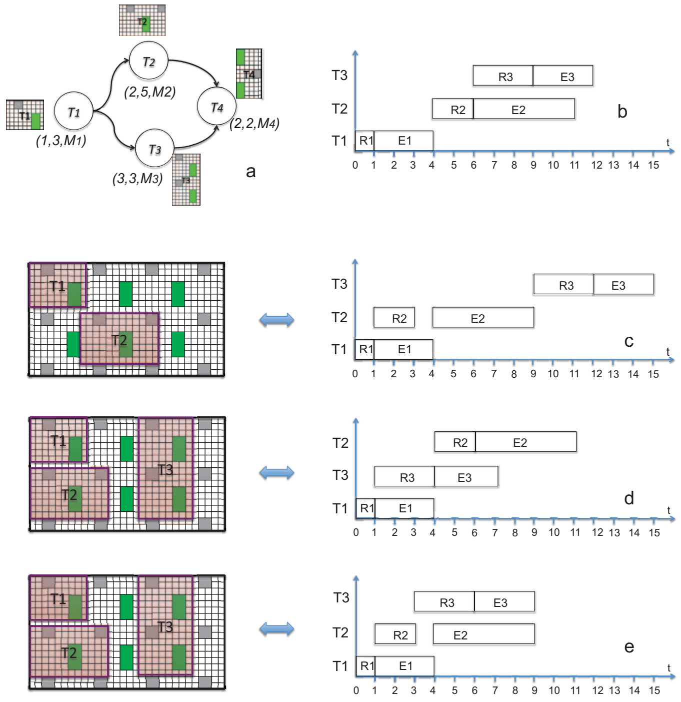

Fig. 1. -a-Example of task graph, parameters under each task mean (reconfiguration time, execution time and model of the task) -b, c, d, e-Different scenarios of task scheduling and placement for the task graph

Despite the advantages offered by prefetching, this technique could be inefficient if it does not combine with the scheduler at run time to attribute correctly the time for loading tasks and the region to place tasks. The ineffective order of tasks to be loaded could increase the duration of overall execution time. Moreover, due to an ineffective placement decision of a task, there is a possibility that other following tasks ready to be scheduled could not find regions to be placed, therefore they must wait until feasible regions are available. To prove the importance of scheduling and placement in reconfiguration overhead, an example with different scheduling solutions for the task graph in Fig 1.a is studied. Fig 1.b shows the case when prefetching technique is not used. Fig 1.c considers the prefetching but not the task placement. The placement of T_2 prevents T_3, performing T_3 cannot be loaded immediately after the reconfiguration of T_2. After T_1 finishes its execution and is removed from FPGA, T_3 can still not start. Fig 1.d shows an example of prefetching with consideration of placement for T_2 and T_3 but an ineffective start of loading time. If T_3 is loaded in advance T_2, the overall execution time will be longer than in Fig 1.e case when T_2 is loaded before. Therefore, the order of tasks to be loaded and the placement decision are extremely important parameters to reduce the reconfiguration overhead.

Finding the best solution in terms of overall execution time is an NP-hard problem when task scheduling, placement and configuration prefetching all need to be considered at one time. Different decisions of scheduling and placement could produce different solutions of overall execution time. The effort to find the best solution by seeking all the possibilities can not be solvable in polynomial time. Thus, it is necessary to develop a run-time spatio-temporal scheduling heuristic which is able to give a good solution in a short time. To achieve it, in this paper, we propose a new heuristic for run-time scheduling and placement of tasks on a 2D heterogeneous eFPGA layer in order to minimize the overall latency of the application. Our heuristic tries to load tasks as early as possible while considering two factors: the priority of new tasks to be loaded and the placement decision to avoid conflicts between tasks.

The remainder of the paper is organized as follows. The next section introduces the related work. Section 3 describes the architecture model, task model and the objective. Section 4 details our scheduling and placement heuristic. Section 5 presents the experimental results and Section 6 concludes the paper with some perspectives.

II. State of the art

In the literature, several techniques have been addressed to reduce the reconfiguration overhead, such as: configuration compression [3], configuration caching [4] and configuration prefetching. Elena perez-ramo et al [5] also proposed a survey of various techniques to reduce the reconfiguration overhead. However, in this paper, we concentrate only on the related works about configuration prefetching.

F. Redaelli et al [6] proposed an exact ILP formulation for the task scheduling problem on 2D homogeneous reconfigurable FPGAs in order to minimize the overall latency of the application. They proposed the Napoleon heuristic based on prefetching technique that reaches a good solution in a short time but the heuristic didn't allow finding a feasible schedule and mapping at runtime. All decisions are taken at design-time, the heuristic must know in advance the task graph in order to compute and sort the task set in increasing order of ALAP values before starting the scheduling and placement step. It is not always the case for applications where the behavior of tasks can be changed at run time. Moreover, for complicated task graphs with many dependent tasks, Napoleon heuristic could take a significant time. J. Resano et al [1] presented a hybrid prefetch heuristic scheduling the reconfigurations at run time but carring out the scheduling computations at design-time. K. Jiang et al [7] proposed a CLP formulation and a heuristic that schedules prefetches and simultaneously performs HW/SW partitioning in order to reduce the expected execution time of an application. J. Edward [8] et. Al. presented an algorithm which predicts hardware execution and tries to prefetch hardware as early as possible while minimizing the risk of misprediction. The reconfigurable region is a 1D FPGA with n columns of the same size and they consider that the placements of the hardware modules are fixed in advance. Y. Qu et al [9] presented three static bases scheduler using prefetching, the first is developed from list-based schedulers where each task has a priority representing the urgency of the configuration. The second is based on constraint programming and the last uses a guide random search strategy. The target architecture is one-dimensional configuration model. J. Li [10] presented an interesting configuration prefetching approach to reduce the reconfiguration overhead for FPGA. However, the prediction for loading task may be erroneous and the task is loaded with the consequent penalization.

The main difference of our work compared to all previous works is that in our work, the loading time for each task is decided at runtime, we consider that we do not have information about the entire future but only the very near future. Moreover, while previous works target the 1D and/or 2D homogenous FPGA, we address the 2D heterogeneous FPGA which is the case of all most recent FPGAs. This heterogeneity imposes stricter placement constraints for task and requires a different strategy of placement compared to a homogenous FPGA.

III. Model and objective

In this section, we present the platform overview and explain how the accelerators (tasks) are managed on it. Then we present the platform model and the application model. From these models, the problem formalization is presented at the end of this section.

A. Platform description

Our target 2D heterogeneous FPGA is inspired from the eFPGA layer of a 3D stacked chip in the ongoing Flextiles project [11]. The 3D stacked chip is composed of a manycore layer and a reconfigurable (eFPGA) layer as in Fig 2. The manycore layer contains several General Purpose Processor (GPP) and Digital Signal Processor (DSP) cores. The reconfigurable layer, dedicated for hardware accelerators, is used to provide a high level of flexibility by support the feature of dynamic reconfigurable paradigm. The NoC on the manycore layer is used to support the communication between processors, between processors and accelerators, or between accelerators.

In this paper, we focus on the scheduling and placement of dependent accelerators on a 2D heterogeneous reconfigurable architecture. Thus, we simplify the Flextiles architecture in

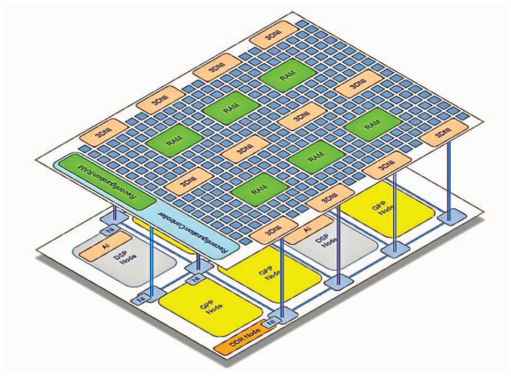

Fig. 2. Flextiles architecture Overview

order to reduce the complexity of the problem. However, this reduction does not affect the mean fullness of our approach but allows to focus on the essential research and evaluation. This architecture is based on a large reconfigurable area connected to one processor via a bus which supports the communication between accelerators, see Fig 3. The processor supports the execution of an Operating System able to control the task management. Furthermore, this architecture contains a shared memory for the storage of data transferred between tasks when needed.

The 2D heterogeneous FPGA has different types of resources, which are symmetrically located at fixed positions on the layer (see Fig 3): computing resources (configurable logic blocks - CLBs), memory resources (Blockram - BRAMs) for storing data during computations and Accelerator Interfaces (AIs) to provide access points between processors and accelerators. Fitting with the reconfigurable technology based on a virtual bitstream, the symmetry of resources on the layer allows a high flexibility in terms of accelerator relocation and increases the designer degree of freedom.

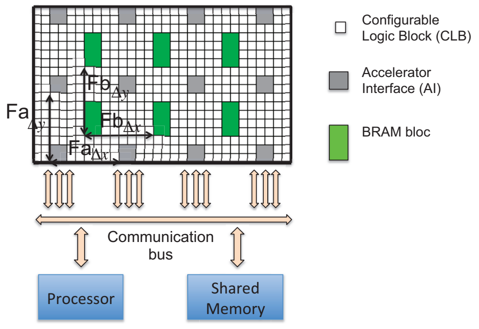

Fig. 3. 2D Heterogenous FPGA

We consider that an accelerator must contain at least one AI and one BRAM. The BRAM is used in each accelerator to receive the data from other accelerators and also to store data during its execution. The AI is used to homogenize the control of accelerators by the processor, but also enables data transfer from/to software and to/from hardware execution. The functionality of the AI is supposed to be the same as in Flextiles and it offers various management services as configuration, control, and debug. Each AI is connected to the communication bus as described in Fig 3. When an accelerator requires more than one AI, one of them must become active and support the communications, others stay inactive. If two accelerators need a data exchange, the operating system must configure two I/O AIs of these two accelerators for setting up the communication. Then, data will be sent from one AI to another AI. The shared memory is needed when one accelerator must send data to another accelerator which has not been configured yet due to some dependencies.

B. Platform model

The execution platform for our problem is an FPGA which is defined by its size (Fw, Fh), with Fw and Fh the width and the height of the FPGA in number of logic elements. Due to the symmetry of the FPGA, the locations of BRAMs (or AIs) are defined by the distance between two adjacent BRAMs (or AIs) but also by the position of the first instance of the block. The horizontal (respectively vertical) distance between two BRAMs is $Fb_{\Delta x}$ (respectively $Fb_{\Delta y}$) and the position of the first BRAM instance is given by Fb_{x0} and Fb_{y0}. The same distances and initial position of first instance are defined for the AI, as $Fa_{\Delta x}$, $Fa_{\Delta y}$, Fa_{x0} and Fa_{y0}. The coordinates $(0, 0)$ corresponds to the bottom left corner of the FPGA.

From these values, the size of FPGA and the location of the different elements can be defined and all symmetrical FPGAs can be supported. An FPGA is then completely defined by

$$\begin{aligned} FPGA &= \{(Fw, Fh), (Fb_{x0}, Fb_{y0}), (Fb_{\Delta x}, Fb_{\Delta y}), \\ &\quad (Fa_{x0}, Fa_{y0}), (Fa_{\Delta x}, Fa_{\Delta y})\} \end{aligned} \tag{1}$$

As an example, the FPGA represented in Fig 3 can be defined as $FPGA = \{(30, 18), (6, 3), (8, 8), (2, 0), (8, 8)\}$

We call Nb_{total} the total amount of BRAMs and Na_{total} the total amount of AIs on the FPGA; Nb_{total} (respectively Na_{total}) is computed by an evaluation of the amount of BRAMs (respectively AIs) which can be placed in the area defined by $Fw * Fh$. This computation is given by

$$\begin{aligned} Nb_{total} &= (\lfloor (Fw - Fb_{x0} - 1) / Fb_{\Delta x} \rfloor + 1) \\ &\quad *(\lfloor (Fh - Fb_{y0} - 1) / Fb_{\Delta y} \rfloor + 1) \end{aligned} \tag{2}$$

Reconfiguration of hardware task on such platform consists in allocating a rectangular zone and loading the bitstream towards the reconfiguration port. A task is supposed relocatable in several regions of the FPGA, the possible regions are defined by the presence of the necessary resources. In this context, as FPGA definition, the model of an hardware task is defined by its size, and the location of each specific block included inside. So this model is defined as

$$M_i = \{(Mw_i, Mh_i), (Mb_{i,x0}, Mb_{i,y0}), (Ma_{i,x0}, Ma_{i,y0})\} \tag{3}$$

with Mw_i and Mh_i are the width and height of the model M_i for a relocatable and reconfigurable task T_i, $(Mb_{i,x0}, Mb_{i,y0})$ the position of the first instance of BRAM in the model M_i, and $(Ma_{i,x0}, Ma_{i,y0})$ the first instance of AI in the model M_i. We define Nb_i as the amount of BRAMs required by task T_i, Na_i the amount of AIs required by task T_i. By applying the same calculation as Nb_{total} and Na_{total} for the task T_i, Nb_i and Na_i could be computed.

C. DAG graph

Similar to some of previous works, we use Directed Acyclic Graph (DAG) to represent dependent tasks. Applications, e.g. image processing, containing a linear sequence of tasks which exchange large amounts of data between them, are examples of this type of graph. DAG is denoted $G = (T, E)$ where T is a set of N_T nodes and E is a set of N_e directed edges. Each node in DAG represents a task T_i $\forall i = 1, \dots, N_T$, while an edge signifies that data produced by one task is used by another one. A task is not ready for scheduling until all its predecessors are finished, we note $Pred(T_i)$ the list of predecessors of tasks which produce data for the task T_i.

As a task is a hardware task, it is necessary to specify not only the execution time ($Trun_i$) for each task but also i) the reconfiguration time ($Trec_i$), and ii) the model for T_i (M_i). The communication time between tasks is not considered since it is taken into account by the execution time. Fig 1.a shows an example of DAG, each task T_i is represented by $(Trec_i, Trun_i, M_i)$. The parameter M_i of each task represents the corresponding model of the task T_i and is illustrated by the small rectangle next to T_i as in Fig 1.a. According to the model definition of task, we can describe M_1 = {(9,7), (6,0), (2,5)}, M_2 = {(12,8), (6,1), (2,6)}, etc.

D. Formalization of the Spatio-temporal scheduling problem

The objective of our algorithm is to minimize the overall execution time of dependent tasks running on 2D heterogeneous FPGA. To achieve that, the scheduler must be able to handle all the following requests:

- when to reconfigure a task (temporal reconfiguration) ?
- where to place the task (spatial reconfiguration) ?
- when to start the execution of a task according with its precedence constraints (temporal scheduling) ?

An ILP formalization for scheduling and placement of tasks on 2D homogenous FPGA is described in [6]. As we consider the heterogeneity of the FPGA, different types of resources and also the placement constraints of tasks on these resources must be taken into account. In this part, we introduce a formalization with the objective function to optimize and the main constraints of task scheduling and placement. We first define some variables as

- Tl_i: time that T_i starts to be loaded in FPGA
- Te_i: time that T_i starts to be executed in FPGA
- $Ta_{i,t} = 1$ if T_i is present on FPGA at time t, 0 otherwise;
- $P_{T_i,R_k} = 1$ if T_i can be placed on the reconfigurable zone R_k, 0 otherwise. R_k is defined below.
- $FR_t = 1$ if the reconfiguration port is free at time t, 0 otherwise. We suppose that only one reconfiguration port is possible, i.e. only one task can be loaded at a time.
- $Nb_{f,t}$: amount of non-utilized BRAMs on FPGA at time t
- $Na_{f,t}$: amount of non-utilized AIs on FPGA at time t
- t_{global} : the overall execution time of the task graph
$$t_{global} = max\left\{(Te_i + Trun_i)\right\}; \qquad \forall\, i = 1, \dots, N_T \quad (4)$$
with N_T the total amount of tasks in the graph

1) Objective: Minimizing the overall execution time of the task graph defined as
$$min\,(t_{global}) \qquad (5)$$
This minimization must respect followings constraints:

2) Placement Constraints: First of all, the minimization must ensure that all the tasks are scheduled one and only one time during the execution of the application. This constraint is defined as
$$\exists\, R_k \mid P_{T_i,R_k} = 1 \qquad \forall i = 1, \dots N_T \qquad (6)$$
with R_k a feasible region for the task T_i, defined as
$$R_k = \{Rx_k, Ry_k, Rx_k + Mw_i, Ry_i + Mh_i\} \qquad (7)$$
with Rx_k and Ry_k the coordinates of the reconfigurable zone R_k. We call R_k a feasible region for T_i when and only when it satisfies three following conditions:

- The amount of BRAMs (respectively AIs) required by T_i must be less than the amount of non-utilized BRAMs (respectively AIs) on the FPGA at time t, for $t = Tl_i$
$$Nb_i < Nb_{f,t} \;\&\; Na_i < Na_{f,t} \qquad (8)$$
$Nb_{f,t}$ and $Na_{f,t}$ are calculated by :
$$Nb_{f,t} = Nb_{total} - \sum_j Nb_j \;\mid Ta_{j,Tl_i} = 1$$
$$Na_{f,t} = Na_{total} - \sum_j Na_j \;\mid Ta_{j,Tl_i} = 1 \qquad (9)$$
- If T_i is placed at coordinate (Rx_k, Ry_k), all resources of T_i must fit on available resources of FPGA. This condition can be expressed by the following constraints
$$\begin{aligned} Rx_k + Mb_{i,x0} &= Fb_{x0} + m_1 * Fb_{\Delta x} \mid \exists\, m_1 \in N \\ Ry_k + Mb_{i,y0} &= Fb_{y0} + m_2 * Fb_{\Delta y} \mid \exists\, m_2 \in N \quad (10) \\ Rx_k + Ma_{i,x0} &= Fa_{x0} + m_3 * Fa_{\Delta x} \mid \exists\, m_3 \in N \\ Ry_k + Ma_{i,y0} &= Fa_{y0} + m_4 * Fa_{\Delta y} \mid \exists\, m_4 \in N \quad (11) \end{aligned}$$
The first line of these constraints means that if the task T_i is placed on the reconfigurable zone R_k, a first BRAM must be found at horizontal relative position $Mb_{i,x0}$, or absolute position $Rx_k + Mb_{i,x0}$. This verification is true if a BRAM exists at this absolute position on the FPGA. We use the symmetric formalization of the FPGA platform to verify if a BRAM is located at a horizontal position $Fb_{x0} + m_1 * Fb_{\Delta x}$. If $m_1 \in N$ exists, it means that a BRAM is present at this location. The same verifications are done for other BRAMs, and also AIs.
- The rectangle $R_k = (Rx_k, Ry_k, Rx_k + Mw_i, Ry_k + Mh_i)$ does not overlap with other tasks currently placed on the FPGA and does not be larger than FPGA size.
$$\begin{aligned} \nexists\, R_l \mid P_{T_j,R_l} = 1 \quad &\wedge \quad Ta_{j,Tl_i} = 1\; \forall j \neq i \qquad (12) \\ &\wedge \quad Rx_k < (Rx_l + Mw_j) \\ &\wedge \quad (Rx_k + Mw_i) > Rx_l \\ &\wedge \quad Ry_k < (Ry_l + Mh_j) \\ &\wedge \quad (Ry_k + Mh_i) > Ry_l \\ &\wedge \quad (0 \leq Rx_k < Rx_k + Mw_i \leq Fw) \\ &\wedge \quad (0 \leq Ry_k < Ry_k + Mh_i \leq Fh) \end{aligned}$$

The first five lines of the constraint defined in equation 12 verifie that it does not exist another rectangle $R_l = (Rx_l, Ry_l, Rx_l + Mw_j, Ry_l + Mh_j)$ currently on the FPGA for which there is an intersection with R_k. The last two lines of the constraint mean R_k must be located inside the FPGA area.

3) Reconfiguration Constraints: A task T_i can be loaded at time $t = Tl_i$ on the FPGA when and only when it satisfies the following conditions:

- At least one feasible position R_k for T_i is found, i.e.

$$\exists\ R_k \mid P_{T_i,R_k} = 1 \tag{13}$$

- The configuration port at time Tl_i is free; i.e

$$FR_{Tl_i} = 0 \tag{14}$$

- All predecessors of T_i finished their reconfigurations, i.e

$$Tl_i \geq max(Tl_j + Trec_j)\ \forall j \mid T_j \in Pred(T_i) \tag{15}$$

4) Execution Constraints: A task T_i can be executed at time $t = Te_i$ on the FPGA when and only when it satisfies the following conditions:

- the reconfiguration of T_i is finished, i.e.

$$Te_i \geq Tl_i + Trec_i \tag{16}$$

- all predecessors of T_i finished their executions, i.e.

$$Te_i \geq max(Te_j + Trun_j)\ \forall j \mid T_j \in Pred(T_i) \tag{17}$$

IV. Scheduling and Placement heuristic

Even if the optimal solution could be found by using Branch and Bound technique that searches for all ILP solutions first, and then takes out the best solution, the effort in time required for the calculation is large. Our heuristic targets runtime applications and does not try to find the best solution but aims at finding a "good" solution in terms of the overall execution time (equation 5) in a short time. As mentioned earlier, in our case, the scheduler does not have information about the entire task graph. Indeed, each task has uniquely the information of its predecessors and its successors. We suppose that these informations are only known by the operating system when the task is scheduled.

The algorithm is based on an heuristic which consists in prefetching tasks as early as possible while considering two factors: the priority of new tasks to be loaded and the placement decision to avoid conflicts between tasks. The detail of our heuristic is given in two parts: Scheduling and Placement. An example is given at the end of this section to ease the understanding of our heuristic.

A. Scheduling

$Algorithm 1$ shows the pseudocode of our heuristic. It is related to the spatio-temporal scheduling by determining when and where to reconfigure a task and when a task can be executed. We list here the parameters used in the pseudocode:

- t: current tick time of the operating system. t is related to the time when a task starts (or finishes) its reconfiguration or its execution
- $timeList$: list of the tick time t for the operating system
- $currentTasks$: list of tasks present on FPGA at time t
- TL : list of tasks available to be scheduled at time t
- L : list of next available successors of T_i whose predecessors (except T_i) have all been scheduled and allocated
- PL: list containing tasks available to be scheduled at time t and next available successors of T_i. PL is the fusion list of TL (without T_i) and L
- $FPGA$, FR, $Trec_i$, $Trun_i$, Te_i, R_k : parameters defined in the previous section

These parameters will be updated at each time t.

Algorithm 1 Scheduling and Placement heuristic

```
1:  function **ScheduleAndPlacement** {
2:  $t$ = 0
3:  $timeList$ = $\emptyset$
4:  $TL$.add(StarterTask)
5:  while $TL \neq \emptyset$ do
6:      Update($FPGA$, $currentTasks$, $FR$)
7:      for all $Ti \in TL$ do
8:          Establish $L$: {Tj = nextAvailableSuccessors(Ti) | Tj $\notin$ TL}
9:          $PL = TL \setminus Ti \cup L$
10:         $R_k \leftarrow$ findFewestConflictRegion($Ti$, $PL$)
11:         if $\exists R_k$ and ($FR$ == 0) then
12:             $FR$ = 1
13:             $FPGA$.load($T_i$)
14:             $currentTasks$.add($T_i$)
15:             $timeList$.add($t$ + $Trec_i$, $Te_i$, $Te_i$ + $Trun_i$)
16:             $TL$.delete($Ti$)
17:             $TL$.add($L$)
18:             Sort $TL$ in descending order of execution time
19:             break;
20:         end if
21:     end for
22:     $timeList$.delete($t$)
23:     $t$ = min($timeList$)
24: end while
25: }
```

The $Algorithm$ 1 works as follows. At the beginning, t equals to 0, $timeList$ is empty and TL contains only the StarterTask (source) of the DAG task graph (line [2-4]). The main part of the algorithm is presented from line [5-24] detailing what happens at each time step t. It is repeated until TL is empty, i.e. all the tasks have been scheduled.

The function $Update$ checks whether a task finishing the reconfiguration or the execution at current time t and then updates the $FPGA$ state, the $currentTasks$ list and the reconfigurator port FR state. FR is busy during the reconfiguration phrase of a task and becomes free when a task finished its reconfiguration. When a task finished its execution, it will be removed from the $FPGA$ and also from the $currentTasks$ list.

After the updating phase, the scheduler will schedule and allocate tasks of TL one by one. The order of tasks to be scheduled follows their order in TL. At time t when T_i is scheduled, the heuristic tries to find the feasible region R_k for placing T_i onto FPGA in order to favor the placement of other remaining available tasks in TL and also next available successors of T_i. This region R_k found by $findFewestConflictRegion(Ti, PL)$ function, must satisfy the conditions for the placement constraints mentioned in the formalization part. The details about $findFewestConflictRegion(Ti, PL)$ will be specified in the next section. If R_k for T_i exists (condition 13) and the reconfigurator FR is free (condition 14), T_i will be loaded into the FPGA. Consequently, $timeList$ and $currentTasks$ will be updated, available successors of T_i will also be added to TL. Then, to reduce the reconfiguration overhead, tasks in TL will be sorted in descending order of their execution time.

Finally, we jump to the next tick time by performing the minimal value in $timeList$ (line [22-23]) and continue to schedule next task in TL until TL is empty.

B. Placement

This section details the technique used in the function $findFewestConflictRegion(T_i, PL)$ called by algorithm 1. The technique aims to find the most suitable position for T_i at time t when it is scheduled, can be divided into two parts: the first part called Fast Feasible Region Search serves to find quickly all feasible regions for the task T_i at time t on the FPGA, the second part called Avoiding Conflict Technique allows to evaluate all the feasible regions for T_i, then choose the most suitable one to place T_i.

1) Fast Feasible Region Search: To the best of our knowledge, most of the hardware task placement algorithms deal with 1D or 2D FPGA homogenous architecture, for example KAMER [12], Vertex List [13], etc. However, real FPGAs have BRAM blocks, multipliers and DSPs in a certain disposition and this heterogeneity imposes stricter placement constraints for the task. Few algorithms deal with task placement on 2D heterogeneous architecture. M.Koester [14] proposed a placement algorithm which is able to deal with the constraints of the hardware tasks. However, feasible positions of the task are not found in run-time. For each hardware task, the given set of feasible positions is predefined at design time. Eiche et al. [15] implemented an on-line placer for heterogeneous devices by using a discrete hopfield neuronal network. They consider that the FPGA is divided in several Partial Regions and a task must contain at least one of them. This consideration can create a waste of resources when a task does not need the entire resources in the PRR.

In this part, we propose an efficient method allowing to find quickly all reconfigurable regions R_k on the 2D heterogeneous FPGA where T_i can be matched at time t. We define FP as the set of these feasible regions ($FP = \{R_k\}$). Instead of seeking each logic block along Fw and Fh of the FPGA to find a feasible region R_k, which is very time consuming, we propose to seek directly by the resources having fewest amounts on FPGA. In our case, the amount of BRAMs is less compared to AIs and CLBs, therefore we start scanning with BRAMs. The idea is trying to place the task in the region where the first BRAM of T_i matched with a BRAM of the FPGA and checking whether the region created is a feasible region for the task T_i. Thus, by placing the first BRAM of T_i successively on all BRAMs of the FPGA, all feasible regions could be found. By doing this way, the complexity for searching feasible regions is reduced $(Fw*Fh)/Nb_{total}$ times.

Fig 4.a represents the proposed searching method. Before scanning BRAMs, we realize a first check about the amount of

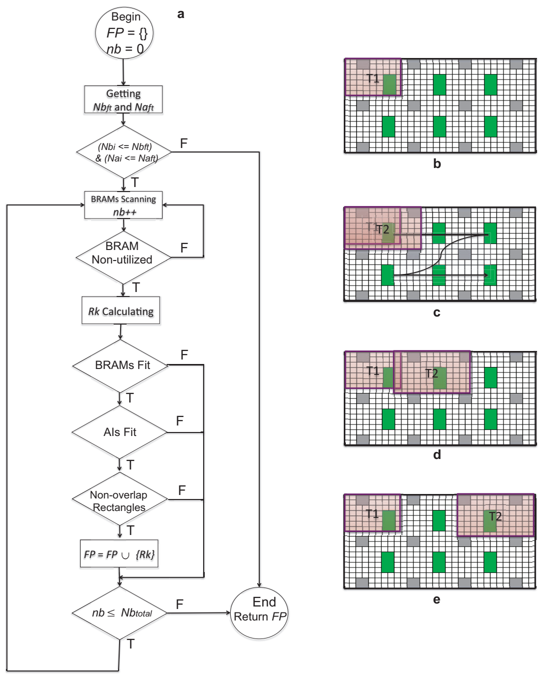

Fig. 4. Quick search method for finding all feasible regions R_k for the task T_i at time t

non-utilized BRAMs ($Nb_{f,t}$) and AIs ($Na_{f,t}$) on the FPGA at time t (condition 8). If the amount of BRAMs and AIs required by the task is superior to the amount of non-utilized BRAMs and AIs on the FPGA, no feasible region is available for T_i. On the contrary, we could start BRAMs scanning process. The BRAMs scanning step consists in scanning each BRAM of the FPGA one by one from the left to the right and then from the top to the bottom as in the figure 4.c. For each scanning iteration nb, if the BRAM is utilized (which is verified by BRAM non-utilized condition), we go to the next iteration. If not, we calculate the region R_k where the first BRAM of T_i can be matched with the BRAM of the FPGA. R_k is a valid feasible region for the task T_i when the conditions 10, 11 and 12 are satisfied. BRAMs Fit Condition is used to check if all other BRAMs of the tasks are matched on the non-utilized BRAMs. If it is the case, the same process is done for AIs Fit condition. When BRAMs Fit condition and AIs Fit condition are satisfied, Non-overlap Rectangles (condition 12) must be respected to ensure that the created region R_k will not overlap with other regions where other tasks are running and/or with the border of the FPGA. Once R_k is a valid feasible region, it is added to FP and we scan the next BRAM until all BRAMs of the FPGA are scanned.

Fig 4.b presents an example with T_1 is currently on the FPGA and we must find all feasible regions for T_2. Fig 4.c shows an invalid region for T_2 with BRAM Non-utilized condition not satisfied, Fig 4.d shows an invalid region with BRAMs Fit and AIs Fit condition satisfied but Non-overlap rectangles condition not satisfied. Fig 4.e shows a valid feasible region for T_2.

2) Avoiding conflicts technique: At time t, once all feasible regions for T_i are found, $findFewestConflictRegion(T_i, PL)$ chooses the most suitable one that creates fewest conflicts with tasks in PL, therefore favors the placement of next available tasks, to place T_i. As mentioned, PL contains the next available tasks and the next successors of T_i. We define parameters p and q which allow to evaluate the conflicts of T_i with tasks in PL.

Firstly, the set of feasible regions for T_i, noted FP is found by using Fast Feasible Region Search presented in the previous part. Then, for each $R_k \in FP$, we calculate the amount of feasible regions for each task in PL if T_i is supposed to be placed at R_k. p is defined as the multiplication of all these amounts, and q as the sum of all these amounts. Thus, for all feasible regions of T_i, the maximum value of p, called $pmax$ and the maximum value of q, call $qmax$ could be determined. p equals 0 means one of the task in PL can not be placed due to the feasible position if T_i is placed on. This task must wait until the extraction of T_i or other tasks to have the chance to be placed, thus delaying the execution of the following tasks. The more larger p is, the more the tasks in PL have the chance to be placed simultaneously on the FPGA, thus the more the reconfiguration overhead is reduced.

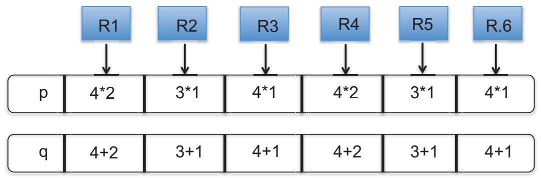

Fig. 5. Calculation of parameters p and q to decide the placement of T_1 in order to avoid conflicts with T_2 and T_3

Our placement decision of Ti is based on the region giving the value of $pmax$ in the case $pmax$ is different to 0. If there is only one feasible region for T_i giving this value of $pmax$, Ti will be placed at this feasible region. In the case many feasible regions are possible, we will take the first furthest feasible region from the center. If $pmax$ is equal to 0, we will take the region giving $qmax$ to finally place T_i. The more larger q is, the more choices to place tasks in PL we have. In the case many feasible regions for T_i give the same $qmax$, we will take the first furthest feasible region from the center.

Fig 5 shows an example how to choose the suitable region to place T_1 of the task graph in Fig 1.a. By using the quick search method, six feasible regions for T_1 denoted R_1, R_2, etc. are found. According to the task graph and the scheduling algorithm, PL contains next available successors of T_1 which are T_2 and T_3. For each feasible region of T_1, p and q are calculated. For example, for R_1, 4 feasible regions for T_2 and 2 feasible regions for T_3 are found, etc. Finally, seeing that $pmax$ is equal to 8, R_1 will be chosen to place T_1.

C. Example

Fig 6 shows by time steps the scheduling and placement of tasks defined in DAG of Fig 1.a. Three different colors are used to differentiate the reconfiguration phase, the execution phase or pending the execution of a task. Next available successors of T_1 are T_2 and T_3, thus T_1 is placed as in Fig 6 at time 0 to favor the placement of T_2 and T_3. T_1 spends 1 time unit to reconfigure. At time 1, T_1 finished its reconfiguration and

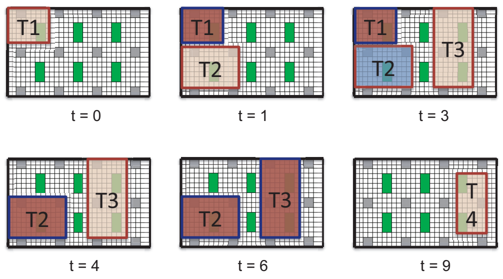

Fig. 6. Scheduling and placement of tasks on 2D heterogenous FPGA

starts its execution. At that time, the reconfigurator becomes free and T_2 can start the reconfiguration. The placement of T_2 must favor the placement of next available task which is T_3, therefore T_2 is placed as in the figure. At time 3, T_2 finished its reconfiguration but it cannot start the execution phase due to the unfinished execution of T_1. The reconfigurator is free at time 3, thus T_3 can start its reconfiguration. The placement of T_3 must favor the next available task T_4. However, no available feasible region for T_4 is found wherever T_3 is placed on the FPGA. At time 4, T_1 finished its execution and is removed from the FPGA, T_2 starts its execution. At time 6, T_3 finished its reconfiguration and starts its execution. Finally, at time 9, T_2 and T_3 are finished their executions and are removed from the FPGA. T_4 can now be loaded. T_4 is placed far from the center as in the figure to favor the placement of next available tasks. In the example, T_4 is the last task to be scheduled, thus there are no available tasks after T_4. After the time 9, T_4 will continue the reconfiguration and execution. At time 13, T_4 finished its execution and is removed from the FPGA. We mark 13 as the overall execution time for the task graph.

V. Results

In order to evaluate the quality of our proposed heuristic, we generate different examples of DAG graphs and compare the results produced by our heuristic to other techniques. The amount of tasks N_T ranges from 5 to 14 tasks for each graph. In each graph, except the source task, one task has at least one predecessor. The reconfiguration time and the execution time range from 1 to 10 time units for each task, the width and the height of each task vary in the interval [6, 18]. A task must contain at least one AI and one BRAM.

N_T	**FFWOP**	**RFWP**	**Napoleon_ex**	**Proposed**
5	41	39.06	39	33
6	48.06	39	39	33
7	60.43	54	54	53
8	56.5	41.63	41	40
9	62.88	48	48	44
10	68.41	52.18	51	51
11	79.84	68	68	66
12	69.88	56	56	57
13	74.98	58.48	58	58
14	102.7	87	87	84

TABLE I

Comparisons of the overall execution time for different techniques

To offer more flexibility for task placement, we simulate the bigger FPGA described as FPGA = {(36,34), (6,3), (8,8), (2,0), (8,8)}. We compare our proposed heuristic with three different techniques: i) First Fit Without Prefetching (FFWOP), ii)

Random Fit With Prefetching (RFWP) and iii) Napoleon (Napoleon_ex). The FFWOP algorithm does not consider the prefetching technique and the order of tasks to be loaded into the FPGA. In FFWP, tasks are placed at the first available region. RRWP considers the prefetching technique but not the order of tasks. In RRWP, tasks are placed randomly on the FPGA at one of its feasible positions. Napoleon's heuristic [6] is very close to our heuristic by considering the prefetching technique and also the task placement. Compared to the Napoleon, our heuristic has two different points: i) While Napoleon algorithm computes the order of tasks to be scheduled at design-time, our algorithm attributes the priority for the tasks at run-time. ii) The placement in Napoleon algorithm addresses the 2D homogenous FPGA and it uses furthest placement criteria in order to increase the probability of placing large modules quickly. In our heuristic, the placement takes care of 2D heterogeneous FPGA and the new task will be placed in the region creating fewest conflicts with other tasks. To compare our heuristic with Napoleon, we extend the basic Napoleon to take into account of the heterogeneity of the task and the FPGA.

The table I shows the comparisons of the overall execution time produced by our proposed heuristic with FFWOP, RFWP and Napoleon extension. In this table, 10 examples of task graph are analyzed. Because FFWOP, RFWP and Napoleon do not consider the order of tasks to be loaded into the FPGA, thus at time t when several tasks are ready to be scheduled, a task is chosen randomly among these tasks to be scheduled. In our examples, for each task graph, FFWOP, RFWP and Napoleon are run 100 times each with the order of tasks is randomly chosen every time. The values produced by FFWOP, RFWP and Napoleon in the table I are the average overall time of these 100 times. By performing the random order of tasks, we can compare fairly our heuristic with others.

For almost cases, our heuristic gives the shortest overall execution time. According to the table I, our heuristic considering the order of tasks to be loaded and the avoid conflict placement technique reduces significantly by 22,5% the overall execution time compared to FFWOP and approximately 5% compared to RFWP and Napoleon. In the case with N_T equals 12, the overall execution time produced by our proposed heuristic are slightly greater than RFWP and Napoleon. This difference is due to the fact that the scheduler does not have the entire information of the task graph. The information about the successors of a task is only known when the task starts reconfiguring into the FPGA, therefore at the time t the list of next available tasks contains only tasks which will be ready to be scheduled in the near future from t. Then, our heuristic favors the placement of next available tasks but cannot predict the placement of far future tasks. However, in some case, the placement of the task T_i at the time t can impact on the placement of T_j that was not in the next available task list at time t.

VI. Conclusion

In this work, we introduce a heuristic to minimize the overall execution time of dependent tasks executed on 2D heterogeneous FPGA. We present as well a technique allowing to search quickly feasible positions of the task on the FPGA. The results show that our proposed heuristic reduces significantly the overall execution time compared to some non-prefetching and other prefetching methods. This work will be extended by taking into account : i) the reused module for a hardware task and ii) the software executions, thus a task can exist in hardware and/or software. An efficient scheduler must be proposed in order to give a good solution in term of the overall time of the application.

References

[1] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1.* IEEE Computer Society, 2005, pp. 106–111.

[2] C. Claus, F. Altenried, and W. Stechele, "Dynamic partial reconfiguration of xilinx fpgas lets system adapt on the fly," *Xcell journal*, pp. 18–23, 2010.

[3] Z. Li and S. Hauck, "Configuration compression for virtex fpgas," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on.* IEEE, 2001, pp. 147–159.

[4] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on.* IEEE, 2000, pp. 22–36.

[5] E. P. Ramo, J. Resano, D. Mozos, and F. Catthoor, "Reducing the reconfiguration overhead: a survey of techniques." in *ERSA*, 2007, pp. 191–194.

[6] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," *International Journal of Reconfigurable Computing*, vol. 2009, p. 7, 2009.

[7] K. Jiang, P. Eles, and Z. Peng, "Co-design techniques for distributed real-time embedded systems with communication security constraints," in *Design, Automation &amp; Test in Europe Conference &amp; Exhibition (DATE), 2012.* IEEE, 2012, pp. 947–952.

[8] J. E. Sim, W.-F. Wong, G. Walla, T. Ziermann, and J. Teich, "Interprocedural placement-aware configuration prefetching for fpga-based systems," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on.* IEEE, 2010, pp. 179–182.

[9] Y. Qu, J.-P. Soininen, and J. Nurmi, "A parallel configuration model for reducing the run-time reconfiguration overhead," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings.* European Design and Automation Association, 2006, pp. 965–969.

[10] Z. Li, "Configuration management techniques for reconfigurable computing," Ph.D. dissertation, Citeseer, 2002.

[11] F. Lemonnier, P. Millet, G. M. Almeida, M. Hubner, J. Becker, S. Pillement, O. Sentieys, M. Koedam, S. Sinha, K. Goossens *et al.*, "Towards future adaptive multiprocessor systems-on-chip: an innovative approach for flexible architectures," in *Embedded Computer Systems (SAMOS), 2012 International Conference on.* IEEE, 2012, pp. 228–235.

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

[13] J. Tabero, J. Septién, H. Mecha, and D. Mozos, "Allocation heuristics and defragmentation measures for reconfigurable systems management," *Integration, the VLSI Journal*, vol. 41, no. 2, pp. 281–296, 2008.

[14] M. Koester, M. Porrmann, and H. Kalte, "Task placement for heterogeneous reconfigurable architectures," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on.* IEEE, 2005, pp. 43–50.

[15] A. Eiche, D. Chillet, S. Pillement, and O. Sentieys, "Task placement for dynamic and partial reconfigurable architecture," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on.* IEEE, 2010, pp. 228–234.