Autoreloc: Automated Design Flow for Bitstream Relocation on Xilinx FPGAs

André Lalevee, Pierre-Henri Horrein, Matthieu Arzel, Michael Hübner, Sandrine Vaton

AutoReloc: Automated design flow for bitstream relocation on Xilinx FPGAs

André Lalevée*, Pierre-Henri Horrein*, Matthieu Arzel*, Michael Hübner†, Sandrine Vaton*
* Telecom Bretagne, Brest, France
{andre.lalevee, ph.horrein, matthieu.arzel, sandrine.vaton}@telecom-bretagne.eu
† Ruhr-Universität Bochum - RUB , Bochum, Germany
michael.huebner@ruhr-uni-bochum.de

***Abstract*—Dynamic and partial reconfiguration of Field Programmable Gate Arrays (FPGA) enable to reuse logic resources for several applications which are scheduled in a sequential order or which are loaded on demand. A fraction of the design on the FPGA is then substituted by another logic function while the rest of the system on the chip stays unaffected. If a design provides several partial reconfigurable areas, the configuration bitstream representing the logic function to be configured in this region has to be adapted to the physical requirements of this chip area. This can be achieved by deploying a repository with all possible configuration bitstreams for all possible regions. It is obvious that storage space can quickly become a limiting parameter in reconfigurable designs. For this purpose, bitstream relocation provides a less storage greedy approach. Only one representation as bitstream of an application needs to be stored. During the configuration process, a relocation algorithm manipulates the bitstream in order to suit it to the respective reconfigurable area. However, reconfigurable regions have to fulfill strong constraints for a relocation to be possible, which makes the selection and placement of reconfigurable regions a complex process. Unfortunately this is not automated by tools so far. In this paper, an approach to automate the development of such relocatable bitstreams is presented along with new algorithms related to relocation specific steps. This approach results in functional designs with minimal intervention from the designer.**

I. Introduction

Dynamic Partial Reconfiguration (DPR) has now grown into a reliable way to enhance flexibility on FPGA designs. Indeed, it offers the ability to modify the functionality of predefined regions of the FPGA during run-time. This provides time sharing capabilities, since a same region of the FPGA can be used for different computing modules at different times. It also increases adaptability of FPGA-based implementations, since required modules can be selected at run-time. Predefined regions can be reconfigured using partial bitstreams, which contain information relative to the reconfigurable resources inside the considered region only.

However, this comes along with several major drawbacks. First, reconfigurable designs usually require more skills and knowledge of the specific target, as the designer has to manually constraint the EDA tools in order to create some regions on the FPGA that would be able to be reconfigured. Second, depending on the design, many modules might be required in many regions. Partial bitstreams contain the implementation of a specific module on a selected region, which means that for each module/region pair, a partial bitstream must be provided. The number of partial bitstreams to store in an application can increase very quickly, as well as the time needed to implement and generate all bitstreams.

Bitstream relocation is a technique that allows a designer to use only one partial bitstream to configure a specific module in multiple distinct regions. This technique aims at selecting similar predefined reconfigurable regions in the FPGA, in order to have the same reconfiguration information for each region. Thus, both the amount of memory needed to store all partial bitstreams and the time required to generate the whole design are greatly decreased. However, this technique requires even more skills and target knowledge than usual dynamic partial reconfiguration in order to be performed, because the user has to identify compatible regions for relocation, as well as manipulate low-level bitstream information. Also, the additional steps required to make a design that allows bitstream relocation are usually time-consuming and error-prone. As a result, relocation is usually too complex to use, despite its promising advantages.

In order to automate some steps of reconfigurable design when using bitstream relocation, some techniques have already been proposed. However, two main issues remained unresolved: floorplanning and timing constraints management. In this article, we introduce a new automated design flow that allows a designer to easily make use of bitstream relocation in Xilinx FPGA. Xilinx is the main FPGA vendor when dealing with DPR. This design flow only requires a small number of parameters from the user, such as the number of required reconfigurable regions. It is partly based on automated scripts for state-of-the-art techniques available in the literature. It also integrates efficient solutions to solve floorplanning and timing constraints issues. The complete design flow is freely available under an open source license [1].

This paper is divided as follows. Section II presents related work on bitstream relocation as well as missing steps required to integrate this technique in an automated design flow. Our proposed design flow, along with the techniques we introduce to perform the previously identified missing steps, are detailed in section III. In section IV, the first tests performed on our design flow are presented, along with its current implementation status. Finally, we present our conclusions and future work in section V.

II. Context and objectives

Among all the existing FPGA technologies, SRAM configuration is one of the most common. It uses SRAM as the configuration layer, which allows fast and volatile configuration storage. One of the characteristics of an SRAM is that each element can be addressed independently. This has led to an interesting possibility: dynamic partial reconfiguration

(DPR). DPR allows to reconfigure only a selected part of an FPGA during run-time, meaning that the remaining portions of the FPGA are still in operation.

A classical partially reconfigurable design is divided into two distinct parts: the reconfigurable part and the static part. The usual design approach is to identify physical regions in the FPGA fabric which are adapted to the desired reconfigurable modules. All the regions which are not selected are used for the static part, and will host the parts of the design which will not be allowed to be reconfigured during execution. Each dynamic module can be hosted in a reconfigurable region for which it has been implemented using a partial bitstream. In the Xilinx PlanAhead design flow [2], the interfaces are placed on fixed locations (*Partition Pins*) in order to be sure that all the reconfigurable regions will have the same *IOs* (Inputs/Outputs).

However, it can happen that some modules have to be configured in more than one region. For each of these modules, one partial bitstream per reconfigurable region is needed. For example, if M modules have to be configured in N regions, $M \times N$ partial bitstreams would be needed. This has several drawbacks. It means long implementation times, since synthesis, placing, routing and bitstream generation are usually long processes. It also increases the bitstream storage place required, and the complexity of run-time reconfiguration management. Storage space requirements can be decreased using data compression algorithms [3]. Very high compression ratio can be obtained (up to 80% according to the experiments in [3]). However, it does not decrease the number of bitstreams to manage, and when dealing with high values of M and N, the resulting space can still be big.

A. Bitstream relocation on Xilinx FPGAs

Bitstream relocation is a technique that allows a partial bitstream to be used to reconfigure a part of the FPGA for which it was not generated. Through this technique, only a single partial bitstream is needed in order to implement a dynamic module in multiple reconfigurable regions. This can result in shorter design times, as each dynamic module only needs one physical implementation regardless of the number of relocatable regions. It can also reduce the memory space required to store partial bitstreams, as only one bitstream is enough to implement it in any relocatable region. Since the use of fast but expensive memories is likely to improve performance, this last point can be a key aspect in reconfigurable designs.

However, bitstream relocation is not as straightforward as a simple reconfigurable design. Indeed, several additional requirements have to be satisfied in order for relocation to be possible. Most of those requirements for Xilinx FPGAs have been detailed in [4] and [5], as well as various ways to fulfill them. The main requirements presented in these papers are:

- identical (size and resources arrangement) origin and destination region
- identical relative partition pins placement
- identical routing between the static part and relocatable regions

The last condition is fulfilled by 1) preventing the static part from using resources located in a reconfigurable region and 2) adding LUTs next to every partition pin.

Once the partial bitstreams are generated, a relocation can be performed by changing the *FAR* (Frame Address Register, *i.e.* the starting configuration address) values(s) in the bitstream. Hardware versions, called REPLICA and REPLICA2, of this address manipulation are provided for Virtex, Virtex-E, Virtex-II and Virtex-II Pro in [6] and [7] though it seems that no hardware relocator is available for newer targets.

However, though recent work [4] [5] includes techniques that ensure the possibility of bitstream relocation, no effort that we are aware of has been presented toward automation of this process, which can be long, error prone, and which may have to be done again manually when a new module has to be implemented (as the predefined regions may not be able to host that new module). Relocation also usually requires extensive knowledge of FPGAs. Moreover, several key steps are missing, such as an efficient automated floorplanning algorithm adapted to bitstream relocation, and a timing constraining technique that can ensure that the relocated module will still respect timing constraints without any functional disorder.

While GoAhead [8] supports bitstream relocation, its approach does not ensure that relocation will be possible. Indeed, instead of making sure that a specific region will be able to host a module generated for another region, this design flow checks if a relocation is possible and regenerates another partial bitstream for the new region if not. This results in a fairly limited control over the relocation for the designer.

Other design techniques focus on the relocation between non-identical regions, as [9], however, we decided to only consider relocations between identical regions in the first version of our design flow, as it keeps the floorplanning and regions constraining simpler, even if these techniques could be added in future developments.

B. Automated floorplanning

The main aim of a floorplanning algorithm is to efficiently partition the FPGA layout between a static part, which will always have the same functionality, and dynamic regions, which will be able to host different algorithms during run-time.

Several algorithms have already been proposed for traditional reconfigurable designs, such as in [8], [10], [11], [12], [13], [14], [15]. However, these algorithms usually outputs very compact static part, *i.e.* designs in which reconfigurable regions are located close to each other in the reconfigurable fabric. Since relocation techniques often prevent the static part from using resources inside reconfigurable regions, some space is required between those regions to allow placing and routing in the case when the static part needs to be directly interfaced with dynamic regions. If reconfigurable regions are too close, it may prevent the static part from accessing the reconfigurable regions, which would render them useless. Another approach presented in [16] makes sure that the static part will have enough space to be successfully placed and routed. However, as all regions have to be identical, this recursive bipartionning approach seems inadequate, since some potential candidates are likely to be discarded if placed on the cuts.

As a result, existing algorithms are not suited to floorplanning for reconfiguration with relocation. Moreover, those algorithms do not take advantage of the fact that all the relocatable regions have to be identical, which drastically reduces the search space, thus resulting in far shorter computation times.

A simple floorplanning algorithm intended for bitstream relocation is presented in [17], but it stops as soon as a valid floorplan is found, without considering other possible floorplans. The resulting floorplan is not likely to be optimal regarding timing issues and regions shapes.

C. Timing constraining

In this paper, we consider relocations between identical regions only. As a result, we can assume that delays inside a reconfigurable module are dependant on the module only, and not on the region. This means that whatever the destination region, delays will remain unchanged when compared with the ones in the region it was implemented for. However, this assumption does not apply to the interface with the static part. It is likely that the nets that belong to the interface between the static part and the partition pins will not have the same delay depending on the relocatable region. Hence, it is possible that the new delay ($new\_static\_to\_dynamic + inside\_relocatable\_region$) will not satisfy the specified timing constraints.

To the best of our knowledge, this problem has not been addressed before. In this study, we present a solution that ensures that delays inside reconfigurable modules will be compliant with timing constraints whatever the relocatable region. As a result, the proposed framework ensures that the resulting design respects timing constraints whatever the dynamic configuration will be.

III. Proposed approach: automated design flow

In this section, we describe our framework to automate the generation of reconfigurable designs using bitstream relocation. After a general presentation, we detail two steps that, as far as we know, have not been fully addressed before in the case of bitstream relocation. This design flow aims at being user-friendly, as it greatly limits user intervention in the process. As a result, it requires less skills than for a traditional partially reconfigurable design. Once all inputs are provided, a simple configuration script has to be run, before a single building command (based on the standard `make` command) generates the whole design.

A. General View

1) Inputs: In order to use our design flow, a designer has to provide the following files:

- an RTL (VHDL or Verilog) description of the static part, in which all reconfigurable regions must be instantiated as black boxes,
- an RTL (VHDL or Verilog) description of each dynamic module,
- a user constraint file (unlike for usual dynamically reconfigurable designs, reconfigurable regions data must not be specified),
- a user configuration file, in which the following parameters must be specified:
  - the part, package and speed grade of the targeted FPGA,
  - the top module of the project,
  - the number of reconfigurable regions,
  - the list of all dynamic modules these regions must be able to host,

It is interesting to note that there is very little difference when compared with a traditional non-DPR design. Apart from a specific hardware design structure in which static and reconfigurable parts are explicitly provided, and from the number of regions and interfaces, all other inputs are expected when designing for an FPGA.

2) Outputs: Our design flow automatically generates a bitstream for the static part, as well as a single partial bitstream for each module. The partial bitstreams only need a modification on the FAR value to be used for another region, which can be easily and automatically computed from the location information available in the UCF (*User Constraint File*). This modification is done through a simple bitstream manipulation program. A software version of this program is currently available, and a hardware version which can be transparently integrated in the design is under development. This hardware version should improve execution time, since it will be pipelined with data transfer to the reconfiguration controller, and a copy of the relocated bitstream is avoided.

3) Automated design flow: Our automated design flow is illustrated in figure 1. This design flow merges the steps of a usual design flow for partially reconfigurable designs, and the required steps for bitstream relocation. On this figure, the usual steps of the *Xilinx PlanAhead* [2] design flow are highlighted in blue. The steps highlighted in green with dashed lines have already been investigated, but never included in a single automated design flow. Finally, the steps in red with thick lines correspond to the original contributions presented in this paper.

All the steps described in the next paragraphs are automated, and launched by the script. The final user does not have to do it himself.

First, the static part and the dynamic modules are synthesized separately using *Xilinx XST*.

The dynamic netlists are then passed to a floorplanning algorithm (see section III-B), which will automatically find the shape and positions of all relocatable regions, based on the resources that are needed by the dynamic modules, and on the number of relocatable regions required by the user. Once the floorplanning is done, a simple shell script writes the constraints related to bitstream relocation in the UCF, accordingly to the regions previously identified, and based on the flow detailed in [4] and [5]. In order to prevent the static part from using resources inside the relocatable region, this design flow uses the *PlanAhead* PRIVATE constraint, which has no equivalent in the *Vivado* design suite yet, which is why our design is still not compliant with *Vivado*.

Then, the static part is implemented using the *PlanAhead* flow. The added LUTs, described in [4], are also placed during this step. The static placed and routed part of the design is analyzed by *FPGA Editor* in order to find the maximum usable

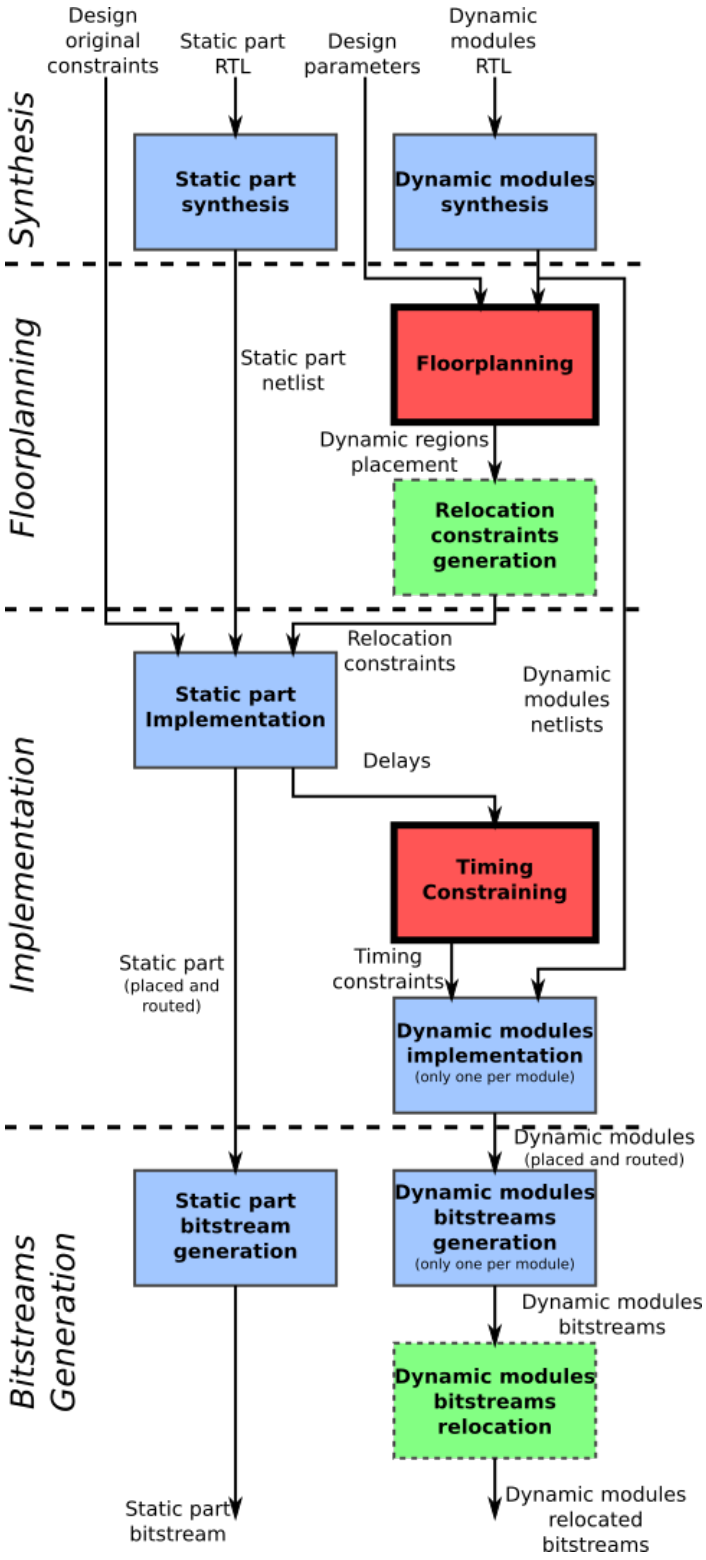

Fig. 1. General view of the proposed design flow

delay for each LUT of the dynamic regions interfaces (more details are provided in section III-C). The dynamic modules are then implemented in only one region, using the timing constraints found by the timing constraining step.

Once the static part as well as all the dynamic modules have been implemented, the tool generates one bitstream per module, which can then be implemented in another relocatable region using a relocation software that simply changes the FAR value(s) in the original bitstream.

B. A new floorplanning algorithm

In order to automatically find a valid placement for all the relocatable regions, a new floorplanning algorithm has been developed. As all the regions must be identical in terms of shape, and resources arrangement, this algorithm is divided into two distinct steps: pattern (*i.e.* shape and arrangement in resources) choice, and regions selection.

1) Pattern choice: The first step of the proposed floorplanning algorithm consists in finding a pattern which will contain enough resources to host each dynamic module and whose number of occurrences on the target is at least equal to the number of reconfigurable regions required by the user.

On a Xilinx FPGA, the reconfigurable fabric is divided into clock regions, which are groups of resources connected to the same dedicated clock network, and that have all the same height. The smaller reconfigurable element is a frame, which is always a clock region high. Usually, if a reconfigurable region's height is not equal to an integer number of clock regions, there is no problem (assuming the reconfiguration is *glitch free*, and that LUTs inside the region are not used to carry state), as the part of each frame that belongs to the static part will be reconfigured the same way it already was. However, in the case of a bitstream relocation, if a reconfigurable region is not an integer number of clock regions high, chances are that the part of frames around the original region that belongs to the static part are different from the one around the destination region. By forcing all the relocatable regions to span the whole height of clock regions, *i.e.* ensuring that each frame will belong entirely either to the static part or to a relocatable region, we can then prevent the relocation from causing any functional disorder. Thus, we will only consider regions that are an integer number of clock regions high in the exploration.

First, we have to evaluate the number of resources (slices, DSPs and BRAMs) that will be needed to implement each dynamic module in a relocatable region. This is easily done, by analyzing synthesis reports provided by *XST*, and storing the maximum found for each resource type. However, while synthesis results can give good indications of the needed resources, they are not fully reliable. Moreover, it is often needed to reserve more resources than necessary to successfully place and route a module inside a region. Thus, we add 10% on the resources estimated in the synthesis reports in order to avoid possible congestions. In the future, we also plan to introduce an optional iterative implementation flow which will decrease the pattern size as long as all modules can be implemented, or increase it if the first pattern found does not provide enough space to implement all modules.

Possible algorithms to find fitting patterns have already been proposed. One solution described in [18] is to search

for all the patterns that are one clock region high (only). This algorithm searches for patterns and removes all patterns that are included in other patterns. While it provides a lot of possibilities, it involves many unnecessary steps and it is rather complex. Another solution provided in [17] is to search for one fitting pattern with the given resources constraints and to stop. This solution is not sufficient since pattern selection is not possible, but the proposed algorithm is efficient, simple, and simple to improve. We decided to extend this second approach to find all fitting patterns.

Also, on recent Xilinx FPGAs, such as 7 series, interconnect tiles are horizontally added between slices, in order for the clock network to be easily routed. This means that if one end of the interconnect tile is included in a reconfigurable region, the other end must also be included in that region. As a result, each slice column of the fabric can only be placed either on a left border or a right border of a reconfigurable region (of course this problem only occurs for slice columns placed on a region border). This also means that if a slice that contains the left part of an interconnect tile is included in a reconfigurable region, the slice that contains the right part of the same interconnect tile must also be included. Thus, the layout description used for our pattern selection must provide information about the possibility for each slice column to be placed either on a right or left border.

An example of our algorithm is described in figure 2. As in the original algorithm, we start on the top left column (one clock region high) of the layout of the FPGA, and shift it to the right until we find a slice column that can be placed on a left border (step *a*), and we extend the region to the right until it meets all the required resources constraints (step *b*). We re-extend it until we meet a slice column that can be placed on a right border (step *c*). We then shrink the obtained region from the left until one resource type is not sufficient enough anymore (step *d*), and we re-extend it to the left until the next slice column that can be placed on a left border (step *e*). Hence, we are sure that the pattern will not be included in another fitting pattern. Then we shift the starting position of the algorithm from one to the right, starting from the slice column that can be placed on a left border of the last found pattern (step *f*), and we iterate the process, until we reach the right border of the FPGA. We iterate this whole process twice: one time by incrementing the starting row of the FPGA (as some rows of the fabric can differ from the other ones), and the second time by incrementing the height of the pattern by one clock region, until we cover the whole fabric. If at any time a non-reconfigurable column is found, the research is restarted (step *f*) on the column after the non-reconfigurable region.

For each pattern found, the tool counts its number of occurrences on the FPGA, and we eliminate the ones that can not be found at least the number of times specified by the designer without overlapping. Finally, the selected pattern is determined using the following criteria (in that order): least number of rows, most number of occurences, least resource waste. These are still arbitrary criteria, since it is difficult to find a fitting score for patterns. In the future, depending on the results of experimental works being carried out, this criteria will probably be refined in order to obtain designs meeting some user-selected optimization criteria such as area waste, or time performance.

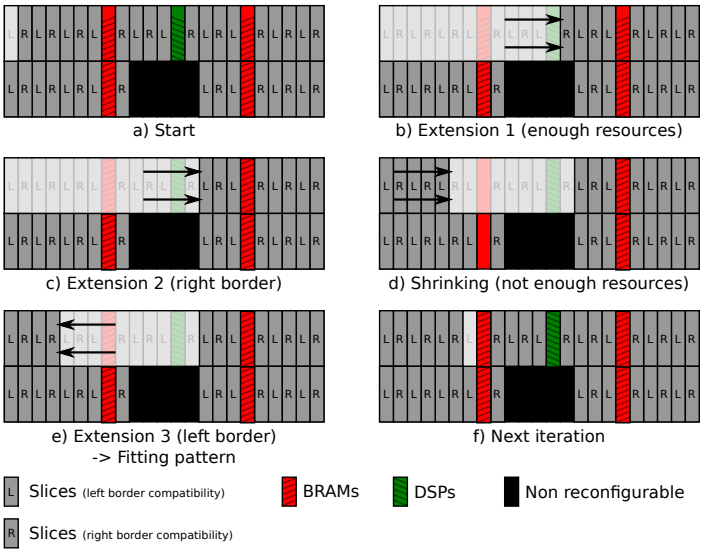

Fig. 2. Example of iterative identification of fitting patterns: needs 8 slice columns, 1 BRAM column and 1 DSP column

2) Regions selection: Once a valid pattern has been selected for our dynamic modules, the tool has to select, among all the occurrences of this pattern on the FPGA, the ones that will be reserved as reconfigurable regions. Of course, configurations where some regions overlap can not be kept.

The resulting selection must respect two criteria. First, the regions must not be too far away from each other, in order to reduce communication delays between them, and thus to improve final performance. Second, the regions must not be too close to each other, in order not to cause congestion problems to the static part that could have to be placed between regions. Indeed, the bitstream relocation prevents the static part from using resources inside reconfigurable regions, so placing those dynamic regions too close to each other could make it impossible for the static part to be successfully (or at least efficiently) placed and routed. This means the algorithm has to identify floorplans with reconfigurable regions close to each other while the static part still has enough space to be routed. One floorplan type that corresponds to these criteria would be a floorplan where all adjacent reconfigurable regions are separated by a distance equal to the minimal number space required by he static part to be routed.

Thus, we had to find an objective function that will meet both criteria when minimized. For each region that belongs to the configuration being estimated, we compute the minimal euclidean distance between each border of that region (4, as reconfigurable regions have to be rectangular) and the closest region (or target border) inside a quarter plan located between two rays starting from the border ends and inclined by +/- 45°from an horizontal line (see an example on figure 3).

For each distance thus obtained, two tests are run:

- if the distance is less than an empirically predefined threshold, a penalty is added to it, equal to $(distance - threshold)^2$ (this way we make sure that regions too close will not minimize the objective function)
- if the distance is the distance with a border of the FPGA, and if it is greater than the threshold, it is not counted

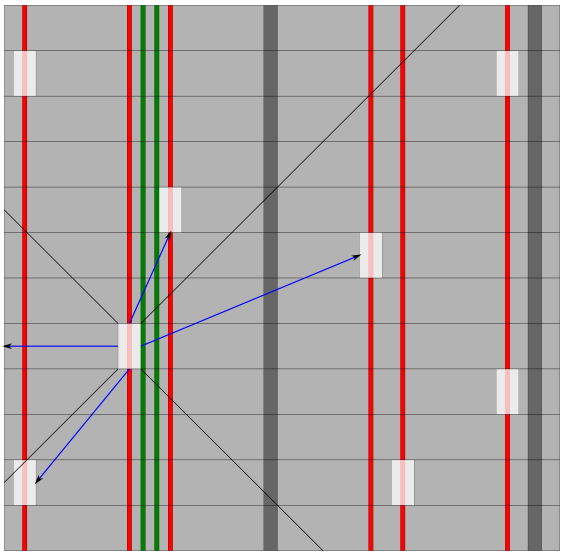

Fig. 3. Example of distances computation for one region (pattern is slice-slice-BRAM-slice-slice × one clock region)

The function we choose to minimize is the sum of the mean of all kept distances and their standard deviation. Though the comparison threshold is still defined empirically (for now equal to the number of slices in a slice column), we plan to refine it based on the static part complexity, which will allow the algorithm to place the regions closer if the static part can be easily routed, and further if the static part is likely to present congestion problems. We also plan to investigate more penalty functions, because while the square function provides satisfying results, some others could be more adapted to this approach.

As the number of possible configurations can be really high, an exhaustive search for the minimization of the objective function is not a viable solution. Hence, we decided to implement a simulated annealing in order to achieve near-optimal results in an acceptable computation time. At each iteration of the algorithm, one region is randomly swapped with another one which did not belong to the previous configuration, and the temperature follows a geometric progression.

Once the simulated annealing is done, the tool just have to write the information relative to the selected regions in the UCF.

C. Timing constraining

When relocating a bitstream without taking any precaution, it is possible that the original timing constraints, which are valid for the origin region, will not be satisfied for the destination region. Indeed, whereas we are sure that both the original and relocated modules will have the same delays inside the reconfigurable regions (as both regions are strictly identical), we can not guarantee that the delays from the static part to the reconfigurable region, or vice-versa, will respect the timing constraints, as we do not have any control over the placement and routing of the static part.

One simple solution to this problem would be to use synchronous interfaces between the static part and relocatable regions, but this would prevent the user from using asynchronous communication protocols (such as a simple req/ack protocol) between the static and dynamic parts.

Instead we propose a new technique that ensures that all the interface delays (*i.e.* from input partition pin to first register, of from last register to output partition pin) inside the reconfigurable regions would be small enough to allow each reconfigurable module to be successfully implemented in any relocatable region of the design.

Once the static part has been implemented using Xilinx *PAR*, the tool uses *FPGA Editor* in order to get all the delays in the design. Then, for each pin that belongs to the interface between a relocatable region and the static part, the tool finds the maximum delay for that pin among all the reconfigurable regions. The maximum delay found is then used to constrain delays for the dynamic modules implementation using the UCF constraints:

PIN "rp_inst_region_number.module_out<*pin_number*>" TPSYNC = regions_output_*pin_number*;

TIMESPEC TS_from_RM_to_PP_output_*pin_number* = TO "regions_output_*pin_number*" (*clock_period* - *max_delay*) ns;

PIN "rp_inst_region_number.module_in<*pin_number*>" TPSYNC = regions_input_*pin_number*;

TIMESPEC TS_from_PP_input_to_RM_*pin_number* = TO "regions_input_*pin_number*" (*clock_period* - *max_delay*) ns;

This way, we make sure that the delays inside each reconfigurable module will be small enough so that this module executes properly inside any relocatable region.

IV. Results and Implementation Status

The proposed design flow has been completely implemented and experimentations have been made to validate its usability.

A. Floorplanning results

Using our floorplanning algorithm, we wanted to know the typical number of regions that we can achieve using bitstream relocation or, on the other end, the typical quantity of each reconfigurable resource that we can use for modules inside relocatable regions. Table I summarizes the maximum number of placeable regions based on the number of needed resources on a Virtex7 690t.

An example result for a design on a Virtex7 690t with 8 relocatable regions, that must contain at least 1000 slices, 8 BRAMs and 6 DSPs, is given in figure 4. From the results, we can see that when using relocation, it is possible to have a good number of reconfigurable regions, as long as the module size stays low. More homogeneous FPGA would have more available regions.

We also ran some tests regarding computing time of our floorplanning algorithm. We found that our algorithm takes 58 seconds to place 15 relocatable regions for a resources need of 1000 slices, 10 BRAMs and 10 DSPs (30 possible placements for each region) on a Virtex7 609t using an Intel Core I5-2500 CPU @3.3GHz. This time is highly acceptable compared to the time it would require to do it manually, and also compared to a usual full design time (synthesis and place & route).

#Slices	#BRAMs	#DSPs	Max #Regions
1000	10	10	30
		40	20
	40	10	10
		40	10
2000	10	10	14
		40	14
	40	10	10
		40	10
3500	10	10	8
		40	8
	40	10	6
		40	6
8000	[0-100]	[0-100]	4
9000	[0-100]	[0-100]	1

TABLE I. MAXIMUM NUMBER OF PLACEABLE REGIONS BASED ON NEEDED RESOURCES ON A VIRTEX7 690T

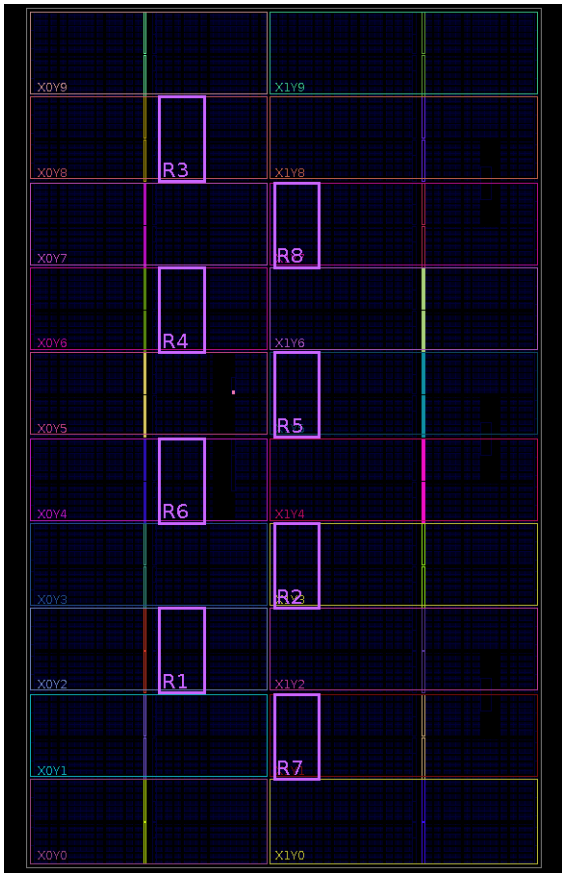

Fig. 4. Example of the result of our floorplanning algorithm (8 regions, 1000 slices, 8 BRAMs, 6 DSPs) on a Virtex7 690t

IP	#Slices	#BRAMs	#DSPs	Max frequency (Mhz)
DFT_8	1048	4	8	542.505
DFT_16	1470	5	12	542.505
Fixed-point_square_root	72	1	24	77.237
Cordic_r_8_8_8	272	0	0	775.964
Cordic_v_8_8_8	305	0	0	475.884
Uniform_Generator	129	0	0	1402.328

TABLE II. RESOURCES REQUIREMENTS AND MAXIMUM ACHIEVABLE FREQUENCY OF THE TESTED RECONFIGURABLE MODULES

B. *Layout description and supported targets*

In order for our floorplanning algorithm to work, we need a description of the layout of the considered target to find patterns and place them on the FPGA. We first considered to use the databases provided by the Torc framework [19]. However these databases do not make the difference between slices M and slices L, which can lead to non-functional relocated modules.

Many Xilinx FPGAs have a vertically homogeneous structure (if we only consider reconfigurable resources), which means that for these targets, only the description of one line (one clock region high) and the number of lines are needed (along of course with parameters that depend on the targeted series, such as the number of each resource in one column). Thus, for those targets, only the horizontal arrangement of resources has to be fully described, as we can consider all non-reconfigurable resources to belong to one type, which can not be included in a reconfigurable region. This presents the main advantage that a new layout can quickly be described and added to the supported targets.

However, some other Xilinx FPGAs present exceptions on their layouts (*i.e.* one line can be different from the others). For these targets, the layout must contain the structure of each row.

This layout description has already been included in the flow and tested on three targets, which are the Virtex5 xc5vlx110t and xc5vlx330, and the Virtex7 xc7vx690t.

C. *Tests*

In order to test the validity of our whole design flow, a more complete test with freely available IPs has been run. The static part makes sure that all the inputs and outputs of the reconfigurable modules are connected, so that no net inside these modules can be trimmed during place and route. This design contains 4 relocatable regions, in which we want to host several modules. The tested modules include two Spiral DFTs (8 and 16 bits) [20], a 32-bit fixed-point square root operator and two cordic operators (8 bits, rotational and vectorial) from OpenCores [21], and a 128 bits 3-tap uniform random number generator presented in [22]. The target used is a Virtex7 690t (speed grade -3). Table II presents the need in resources in all tested modules, as well as the maximum achievable working frequency as estimated by *XST*.

Beside the RTL description of all the reconfigurable modules and the static part, the only information we had to give for our full design flow to properly execute are: the name of the UCF, the number of reconfigurable regions we want in design,

the FPGA target, package and speed grade, and the name of the project top module.

Based on the resources each tested module needs, each relocatable region must contain at least 1617 slices (+10% overhead), 5 BRAMs and 24 DSPs. We fixed the design's clock to 70 MHz, as we wanted a little margin for our slowest tested module (the fixed point square root operator).

Each module has been successfully implemented in one relocatable region and then relocated to the other 3 regions identified by our floorplanning algorithm, with no errors. The same design has also been successfully implemented with the design's clock set to 450 MHz, but this time omitting the fixed point square root operator (as its maximum operating frequency is fairly limiting compared to the other modules).

V. Conclusion and future work

In this paper we presented a new automated design flow for bitstream relocation. This approach can be used to generate designs supporting bitstream relocation without any additionnal intervention from the user compared to usual dynamically reconfigurable designs. This design flow is based on the Xilinx *PlanAhead* design flow, using state-of-the-art techniques already detailed in the literature, and two new algorithms that take care of automated floorplanning and timing constraining adapted for bitstream relocation.

First tests indicate that our approach is functionnal and provides interesting results. Using this design flow requires even less skills than the usual DPR flow, as all floorplanning steps are automated. Adding a new target is straightforward, and the whole design flow is available [1] under an open source license in order to benefit the community.

This project is still under active development. Short term modifications will target refinement of our algorithms in order for them to be able to adapt to the complexity of the design. We also plan to give the user more control over the design flow, such as the possibility to manually define the floorplanning. Next steps also include the possibility to use several patterns simulteanously in the same design, which can lead to less area waste, as small and large modules could use different predefined reconfigurable regions. Longer term improvements include compliancy with the Xilinx *Vivado* suite, and investigation of the possibilities to adapt it to FPGAs provided by other manufacturers.

References

[1] "Autoreloc redmine home page," https://redmine.telecom-bretagne.eu/projects/autoreloc.

[2] [Online]. Available: http://www.xilinx.com/tools/planahead.htm

[3] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, Nov 2004, pp. 766–773.

[4] T. Drahonovsky, M. Rozkovec, and O. Novak, "Relocation of reconfigurable modules on xilinx fpga," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, April 2013, pp. 175–180.

[5] ——, "A highly flexible reconfigurable system on a xilinx fpga," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–6.

[6] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 151b–151b.

[7] H. Kalte and M. Porrmann, "Replica2pro: Task relocation by bitstream manipulation in virtex-ii/pro fpgas," in *Proceedings of the 3rd Conference on Computing Frontiers*, ser. CF '06. New York, NY, USA: ACM, 2006, pp. 403–412. [Online]. Available: http://doi.acm.org/10.1145/1128022.1128045

[8] C. Beckhoff, D. Koch, and J. Torresen, "Go ahead: A partial reconfiguration framework," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 37–44.

[9] T. Becker, W. Luk, and P. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, April 2007, pp. 35–44.

[10] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug 2006, pp. 1–8.

[11] ——, "Physically-aware exploitation of component reuse in a partially reconfigurable architecture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, pp. 8 pp.–.

[12] A. Smith, G. Constantinides, and P. Cheung, "Integrated floorplanning, module-selection, and architecture generation for reconfigurable devices," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 733–744, June 2008.

[13] A. Montone, M. D. Santambrogio, D. Sciuto, and S. O. Memik, "Placement and floorplanning in dynamically reconfigurable fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 24:1–24:34, Nov. 2010. [Online]. Available: http://doi.acm.org/10.1145/1862648.1862654

[14] C. Bolchini, A. Miele, and C. Sandionigi, "Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable fpga systems," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 532–538.

[15] K. Vipin and S. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, O. Choy, R. Cheung, P. Athanas, and K. Sano, Eds. Springer Berlin Heidelberg, 2012, vol. 7199, pp. 13–25.

[16] T. D. Nguyen and A. Kumar, "Prfloor: An automatic floorplanner for partially reconfigurable fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 149–158. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847270

[17] T. Becker, M. Koester, and W. Luk, "Automated placement of reconfigurable regions for relocatable modules," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 3341–3344.

[18] R. Backasch, G. Hempel, S. Werner, S. Groppe, and T. Pionteck, "Identifying homogenous reconfigurable regions in heterogeneous fpgas for module relocation," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–6.

[19] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950425

[20] [Online]. Available: http://www.spiral.net/hardware/dftgen.html

[21] [Online]. Available: http://opencores.org/project,fixed-point-sqrt

[22] D. B. Thomas and W. Luk, "High quality uniform random number generation using lut optimised state-transition matrices," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, pp. 77–92, 2007.