AMAH-Flex: A Modular and Highly Flexible Tool for Generating Relocatable Systems on FPGAs

Najdet Charaf*, Christoph Tietz*, Michael Raitza§, Akash Kumar§, and Diana Göhringer*
*Chair of Adaptive Dynamic Systems, §Chair for Processor Design
Technische Universität Dresden, Germany
E-mail: {najdet.charaf, michael.raitza, diana.goehringer, akash.kumar}@tu-dresden.de, {christoph.tietz}@mailbox.tu-dresden.de

***Abstract*—In this work, we present a solution to a common problem encountered when using FPGAs in dynamic, ever-changing environments. Even when using dynamic function exchange to accommodate changing workloads, partial bitstreams are typically not relocatable. So the runtime environment needs to store all reconfigurable partition/reconfigurable module combinations as separate bitstreams. We present a modular and highly flexible tool (AMAH-Flex) that converts any static and reconfigurable system into a 2 dimensional dynamically relocatable system. It also features a fully automated floorplanning phase, closing the automation gap between synthesis and bitstream relocation. It integrates with the Xilinx Vivado toolchain and supports both FPGA architectures, the 7-Series and the UltraScale+. In addition, AMAH-Flex can be ported to any Xilinx FPGA family, starting with the 7-Series. We demonstrate the functionality of our tool in several reconfiguration scenarios on four different FPGA families and show that AMAH-Flex saves up to 80% of partial bitstreams.**

***Index Terms*—Field programmable gate arrays (FPGA), Floorplanning, bitstream relocation, dynamic partial reconfiguration, dynamic relocatable system**

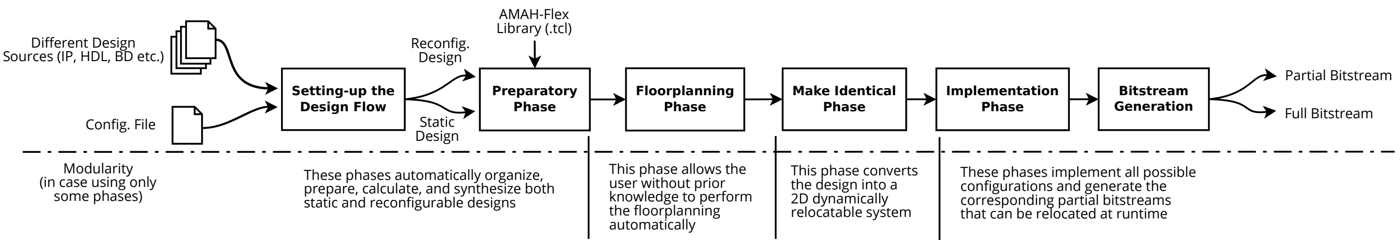
I. Introduction

Field Programmable Gate Arrays (FPGAs) have become an attractive platform for many real-time applications that work in changing environments. The devices themselves deliver a performance that makes them usable for real-world applications. The development of FPGAs has taken hardware flexibility, in general, one step further. In the end, the toolchain to build applications on these devices also has improved significantly, which makes these devices used by a wider engineering audience. Recently, FPGAs are advertised as the "one-takes-it-all" standard embedded platform with the ability to start developing with traditional embedded software design and calling out to specialized hardware accelerators, if needed [1]. Current toolchains steer the development process into the direction of perceiving special hardware call-outs as calls into a software library. And this is, indeed, an effective metaphor. Combined with powerful microprocessors on which the software runs, this leads to the expectation that multiple embedded applications can be consolidated on a single system. Modern operating systems are capable of dynamically managing the required working set of application code and data without the need for detailed planning at development time. The exact working set does not even need to be known at development time, as long as all real-time and power requirements will be met in the end; mixed-critical systems [2] are a well-known representative in this development direction. Two main concepts make the management of software resources transparent to the developer and the user: runtime library loading and virtual addressing; the software simply works, and it easily integrates with other software. This is in strong contrast to the way hardware accelerators are currently handled. They usually still become part of the fixed hardware platform on which the embedded applications run. All resources and all access to shared resources must be planned in advance and immediately thought through to the end. Here is where the metaphor of calling an external library breaks down. The concept of transparent runtime resource management breaks down and almost everything is left for the application developer to solve.

Current toolchains and FPGA designs do not yet support the concept of virtual addressing or bitstream relocation at runtime. Each loaded partial bitstream (PB) represents the functionality of a specific location defined at synthesis time and not at runtime. However, if this location is occupied by another accelerator, we have an unsolvable resource conflict. Besides, it takes a long time to generate all partial bitstream possibilities of a design with several reconfigurable functions. This is very critical for designs with a large number of reconfigurable functions since the storage resources are limited. One approach to overcome these problems is the so-called bitstream relocation [3]. It consists of having a PB that can be manipulated to load into any other compatible reconfigurable region. This technique helps to reduce the memory needed to store all PBs and the time required to generate all PBs. The concept has been around for quite a while. However, there is still a lack of automated and highly flexible tools to transform the current designs into a dynamic relocatable system and make it accessible to a wider audience.

With our proposed work, we present a modular and highly flexible tool that consists of an automated design flow for floorplanning and generating a 2-D relocatable design for different Xilinx FPGAs using Vivado. Moreover, we have closed the remaining gap between synthesis and bitstream generation to relocate partial bitstreams on all Xilinx FPGA families starting from the 7-Series. Our contributions include:

- Accepting different design sources, enabling Isolation Design Flow, and supporting all Xilinx 7-Series and UltraScale+ families (Vivado versions as of 2017.2)
- Automatically find, floorplanning, and place reconfigurable partitions
- A modular and very highly flexible tool that can be adapted, extended, and used separately for individual other purposes



Different Design Sources (IP, HDL, BD etc.)
Config. File
AMAH-Flex Library (.tcl)
Reconfig. Design
Static Design
Setting-up the Design Flow
Preparatory Phase
Floorplanning Phase
Make Identical Phase
Implementation Phase
Bitstream Generation
Partial Bitstream
Full Bitstream
Modularity (in case using only some phases)
These phases automatically organize, prepare, calculate, and synthesize both static and reconfigurable designs
This phase allows the user without prior knowledge to perform the floorplanning automatically
This phase converts the design into a 2D dynamically relocatable system
These phases implement all possible configurations and generate the corresponding partial bitstreams that can be relocated at runtime

Fig. 1: On the upper half a general overview of the design flow of our tool and its phases. On the bottom page, a general description of modularity, in case the subphases are used separately.

This paper is organized as follows: Section II discusses the related work and the comparison with them. Section III and IV give a detailed overview of the proposed tool. Section V summarizes the experimental setup and provides a detailed overview of the applied scenarios. In Section VI, we summarize the paper and give an outlook on future work.

II. State of the Art

In the last years, several works [4]–[8] been presented for placing RPs automatically. They discussed and proposed approaches to efficiently find and place dynamic regions on FPGAs. However, these solutions are not applicable for partitioning FPGAs so that it is possible to relocate bitstreams, since: these works do not fulfill all the major requirements of the bitstream relocation technique that are mentioned and explained in Section III. Further approaches have addressed the integration of bitstream relocation. An automated design flow for bitstream relocation is presented in [9]. This design flow automatically performs the floorplanning phase. The possibility to relocate in 1-D and/or 2-D is not presented. This approach uses ISE and the PlanAhead PRIVATE constraint to prevent the static part from using resources inside the reconfigurable partition (RP). Therefore, the approach can not be used by Vivado Design Suite and is not applicable to newer and future devices. A novel approach to relocate partial bitstreams was presented in [10]. In this work, a technique called RePaBit is developed, which can relocate in a vertical and horizontal direction. RePaBit is a partially automated design flow that uses the Vivado Design Suite. The proposed technique avoids feed-through routes, which are the connections between static resources that cross the RPs, by using Isolation Design Flow (IDF). RePaBit adds two additional look-up-tables (LUTs) to the netlists in both, static and reconfigurable logic, this is called "bus macro". However, using this approach comes along with more LUTs overhead, since for each I/O signal of a reconfigurable module (RM) two LUTs are required, one in the static part and one in the RP. Additionally, the user still has to perform the floorplanning manually and the approach is developed only for Xilinx Zynq FPGAs. Similar to the prior approach an automated design flow using Vivado was presented in [11]. This work proposes a technique using TCL scripts. In order to guarantee a consistent interface between the static logic and the reconfigurable logic, the proposed technique adds an additional partition called Connection Partition (CP). This partition is a complementary partition to each RP for all partition pins to tie to. Unlike the prior approach, this approach inserts for each I/O signal of an RM only one additional LUT, which will be placed in the static part. However, the prevention of feed-through routes and the facilitation to relocate vertically and horizontally is not presented, and like RePaBit, the user still has to perform the floorplanning manually. A newer approach is called IMPRESS [12]. This is a TCL script-based tool that automatically generates bitstreams for bitstream relocation under Vivado. This approach supports IP blocks specially generated with Vivado HLS for the implementation of reconfigurable systems. Besides, this approach enables reconfigurable-to-reconfigurable communication and hierarchical reconfiguration. In the paper, however, it was not clear what the user must provide to the tool and what the tool generates automatically. Furthermore, an automatic floorplanning phase was not presented. Therefore, the user still has to perform this phase manually. Also, the used blocker macro for avoiding static nets within RPs needs a lot of time for realization and cannot be easily transferred to other boards.

Accordingly, and to the best of our knowledge, our proposed tool is the first developed solution that enables 2-D bitstream relocation for all FPGA families of both 7-Series and UltraScale+ architectures, including the Isolation Design Flow (IDF) and using a flexible interface. AMAH-Flex provides the necessary framework to automatically apply the floorplanning phase and transform any system into a dynamic relocatable system. The proposed solution contributes to the current motivation to make reconfigurable systems increasingly flexible in adaptive, ever-changing hardware systems.

III. Requirements and Tool Outline

Our design flow differs from the classic Dynamic Function Exchange Flow by having three layers (two additional layers and one extended layer). The two additional layers: one to build the design flow and one to make all RPs identical so the functions can be relocated at runtime. The extended layer improves the classical synthesis phase by automatically calculating the resource utilization and preparing the design flow for the used architecture. Figure 1 shows a detailed overview of our design flow and describes each phase separately, in case the user uses the sub-phases separately.

A. *Definitions*

In this work, we use the following terminology to describe the parts and relations of our floorplanning and implementation flow.

Dynamic Function eXchange (DFX): The Dynamic Partial Reconfiguration (DPR) technique from Xilinx is now called DFX [13].

Reconfigurable Module (RM): The function to be implemented as a relocatable hardware accelerator.

Reconfigurable Partition (RP): The partition on the FPGA that can hold a certain set of compatible RMs.

Relocation Group (RG): A set of RPs. Each RM that fits into any RP in the relocation group can be relocated to any other RP as well.

AMAH-Flex Library: The framework of the tool which includes all necessary functions for building the 2D relocation design flow.

Configuration File (Config File): Contains the part of the used FPGA, the names of RMs and RPs in the design as well as the names and the paths of all folders.

B. Requirements for relocatable partial bitstreams

Candidate reconfigurable partitions have to meet a set of requirements to be suitable for bitstream relocation. All RPs that are used for bitstream relocation must be identical and fulfill the following characteristics:

Resource footprint: The underlying hardware resources must have the same arrangement.

Interface compatibility: Partition pins and nets which are connected to the partition pins must have the same relative placement, number and routing respectively.

Avoiding of Feed-Through: A static net that crosses a Reconfigurable Partition (without a connection) is called Feed-Through net. These nets are not allowed.

C. Extended Isolation Design Flow (EIDF)

Safety-critical and security-critical systems often have special requirements. In the programmable logic domain, it is the encapsulation of modules to enable independent operation. Xilinx addresses this with their IDF, which allows the isolation of modules on a single FPGA. Since both DFX's baseline and relocation design flow do not actively avoid implementation networks, it is an additional task to implement this feature. Furthermore, there is no toolchain from Vivado that performs automatic floorplanning while preserving IDF rules. As of Vivado version 2020.2, a combination of IDF and DFX is only supported for Zynq UltraSacle+ MPSoC devices [14]. Support for 7-Series FPGAs is not available and will not be in the future according to [15]. For all other UltraScale and UltraScale+ FPGAs, Xilinx may support this combination in future releases [[14], p. 80]. Using AMAH-Flex, it is still possible to fulfill the primary goal of avoiding feed-through paths by switching between these flows during the relocation design flow. With this workaround, it is possible to apply it for different FPGAs. To enable this functionality, the HD.ISOLATED attribute must be set for each hierarchical module that is to be isolated. By restricting Pblocks for each module, isolated regions are created that contain only the logic of their module and the top-level logic. The latter is an important fact: any logic that is not hierarchically within an isolated module is considered top-level logic and can even be placed in isolated regions. Routing of top-level networks can be from, to, or through isolated regions. Communication between isolated modules is implemented with so-called trusted routes. For more information about IDF, see [14].

D. Design Sources and AMAH-Flex Library

To maximize the flexibility of the proposed tool, we implemented an approach that accepts all the different design sources that Vivado supports. For now, AMAH-Flex supports the design sources: *prj, sysvlog, vlog, vhdl, ip, bd, cores* with the file name extensions: *prj, sv, v, vhd, xci, bd, tcl, ngc, edn, edif, edf, dcp*. Other naming extensions can be easily added. Moreover, there are additional options which can be applied for each source. AMAH-Flex supports the design source options: *includes, generics, vlogHeaders, vlogDefines, ipRepo, xdc, synthXDC, implXDC, synth_options, synthCheckpoint*. The library is the framework of the AMAH-Flex tool. It is a TCL script-based library and contains 62 functions. Most of the functions are implemented in a modular and highly flexible way to be used and extended in other projects.

E. Connection Partition

We employ a connection partition (CP) to ease the task of meeting the resource arrangement and routing constraints. The CP allows for decoupling an RP before reconfiguration managed by the static logic. Using a CP infers an additional set of LUTs in the number of I/O connections to the RP. The overhead introduced by connecting to the RP via a separate CP is negligible (see [11]). By decoupling the RP from static logic, it frees the designer to individually implement decoupling themselves. Another advantage is that routes connecting signals that cross the RP boundary are guided towards the CP, further constraining their placement. This makes it easier to find solutions for placing the partition pins but might rarely cause difficulties to find a viable solution for an RP implementation, as it might exhaust routing resources in a particular region of an RP.

IV. Design Flow of AMAH-Flex in Detail

The tool starts reading the information from the *config file* (e.g. Vivado version, FPGA architecture, organization of output folders) and collects all related files into the design. Each different design source has a specific file name extension. This property is used to add the source correctly. A core function called *simple_add_module* remembers all the various design sources that exist in the specified directory. Then for each type of source found, it adds all the files according to their extension to the module attribute. This approach allows us to use mixed design sources. This phase is called *setting up*. The result of this phase is a static design, a reconfigurable design, and setting up all parameters related to the synthesis and implementation process. After this phase, the *preparatory* phase begins.

The *preparatory* phase finds, separates, and stores all RMs as well as the associated FPGA board information and prepares all needed variables for the next phases. The static part and all RMs are synthesized separately. The synthesize process is performed by Vivado. Then, the report results are saved in the *Synth* folder. Another essential step is to collect all necessary information regarding the available resources on the used FPGA

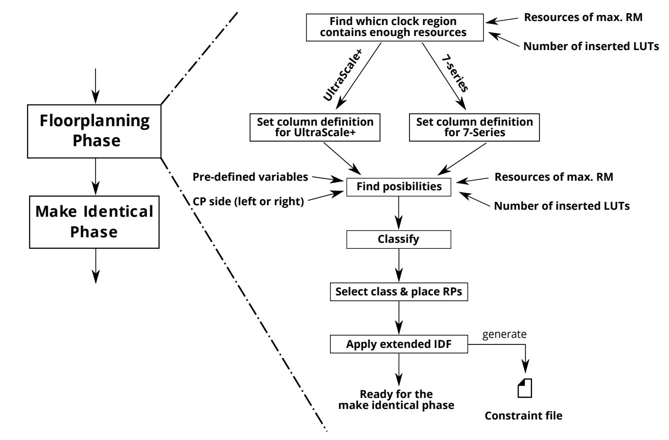


Fig. 2: A general overview of the floorplanning phase.

board. AMAH-Flex writes them to a generated *ARCH file*. The *ARCH file* is generated only once and the step can be skipped if the FPGA information is already available. Once the file is generated, an implemented algorithm sorts and divides its contents into three main groups: Tiles, I/O pins, and clock ranges. Each tile in the tile group is then assigned to a column within a particular clock region. The clock regions are divided into columns and rows, and each column has an individual position number and contains only one type of resource. In this context, it is important to note that the naming of some resource names differs between the UltraScale+ and 7-Series architectures. After the *preparatory* phase was passed successfully, the design flow continues with the *floorplanning* phase.

In general, **the *floorplanning* phase** aims to find a valid placement for all needed RPs on the FPGA, so that it is possible to assign the RMs to them and the communication between the static part and these regions is faultless. The placement of RPs requires extensive knowledge of FPGAs and the common design tools from Xilinx do not take over the task of placing them automatically. That means the user has to manage it manually, which can take a long time and be error-prone. With AMAH-Flex, we offer an automatic floorplanning phase that frees the user from this difficult and demanding task. Figure 2 illustrates the steps of the developed *floorplanning* phase. In this phase, all placement possibilities of RPs on the FPGA are found, which have enough resources to host the largest RM. The *floorplanning* phase begins by checking each clock region if it contains enough resources to host at least the maximum RM plus the related CP. The algorithm then checks whether the FPGA architecture used is a 7-Series or an UltraScale+ architecture. This check is essential because the UltraScale+ (US+) architecture differs from the 7-Series (7S) in terms of elements notation, elements arrangement, and the number of elements within one clock height column (7S: 50 CLBs; US+: 60 CLBs). The algorithm continues with the main sub-phase *find possibilities*, in which all possible positions within each clock region are found. Figure 3 shows the process of the *find possibilities* in details, including the related states for the case where the connection partition is placed on the right side. The *find possibilities* starts with state *Init*, where all relevant parameters are set. *Find possibilities* completes the first stage when all required resources of the

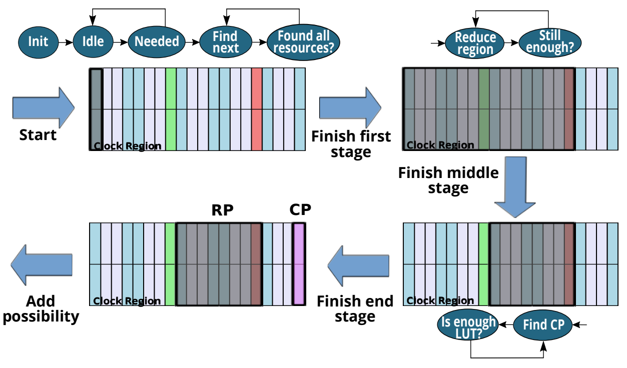


Fig. 3: A detailed overview of the *find possibilities* procedure including the three stages and their states. Example of a placing possibility: 3 CLB columns (blue) and 1 DSP column (red). Gray are the interconnections.

requested RM have been found.

Then, in the middle stage, the RP found is reduced, since resources could be included that are not used because the arrangement of an element type on the FPGA is not successive. In the end stage, the correct CP to its RP is searched, which contains only slices as element type. At the end when the suitable CP is found this combination of RP + CP is saved as a "possibility". This is repeated until all clock regions have been searched. Once all possibilities are found, the design flow algorithm classifies them into separate groups. Each group consists of identical RPs that contain the same resource arrangement as well as identical CPs that contain the same slice type. Further, the distance between the CP from the RP is equal within one relocation group. This step is required, since the design flow proposed in this work considers only relocation between identical resource arrangement within RPs

Algorithm 1: Classify RPs & CPs into relocation groups

```
Classify ();
Input  : All possible positions of RPs + CPs on FPGA
Output : The relocation group array
check IsPossiblePositions not_empty;
forall Clock Regions do
    PrepareAllRelevantVariables;
    StoreRPCPPairwise;
end
while AllPossibilities not_empty do
    switch state do
        case Init do
            SelectNewFootprint($RP_f + CP_f$);
        case Compare do
            AreAllClockRegionThrough;
            AreAllPossibilitiesWithinClockRegionThrough;
            SelectNewPossibility($RP_p + CP_p$);
            if $(RP_f + CP_f)$Equal$(RP_p + CP_p)$ then
                CalculateDistance;
                if DistanceIsEqual then
                    SavePossibilityInTheClassList;
                end
            end
            else
                SavePossibilityInTheTempList;
            end
        case No more entries do
            IsTempListEmpty;
        case Finish do
            GenerateRelocationGroup;
        otherwise do
        end
    end
end
```

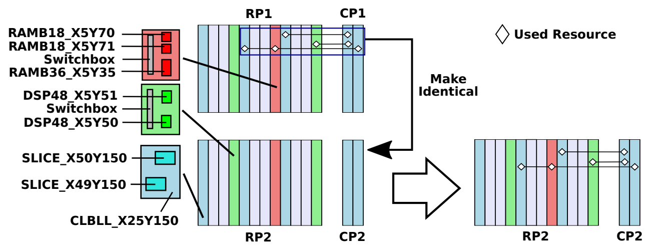


Fig. 4: Example of the result of our *make identical* algorithm on Xilinx Virtex7.

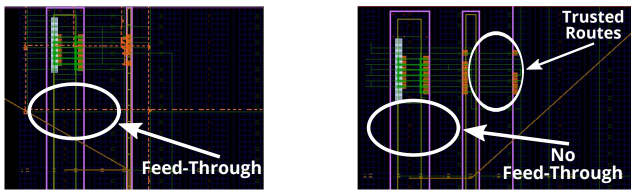


Fig. 5: The final result of AMAH-Flex on UltraScale+. *Left*, an example of a feed-through as an ornage colored dashed arrow (without AMAH-Flex or without IDF). *Right*, using trusted routes and free of feed-throughs (using AMAH-Flex).

and CPs. The procedure for classifying the RPs and CPs into relocation groups is shown in Algorithm 1. Consequently, the class is selected from the classify list, which contains at least the number of resources the user needs for RPs. Afterwards, the RPs are placed as many times as the user requires. Once all RPs and all CPs are successfully placed, a constraint file is generated containing the necessary information needed for the Frame Address Register (FAR) value for each RP. Now the design flow continues with the *extended IDF*. The IDF DRC checker are enabled with *hd.enableIDFDRC = true*. Running the isolation DRC can help to avoid consequential errors. Once the *extended IDF* is successfully completed, AMAH-Flex continues with the *make identical* phase. **The *make identical* phase** makes the content of all RPs identical. This means that, it makes the logical and physical position of the used resources and I/Os as well as the routes between RPs and the static part identical. The aim of this phase is to fulfill all the requirements mentioned and explained in Section III. Figure 4 illustrates the *make identical* phase on Xilinx Virtex7. In this phase the algorithm copies all used resources (e.g. DSP, CLB, BRAM) from the template RP to the target RP as well as from the template CP to the target CP. **The *implementation* phase** routes the static design to all RPs. It also loads all other RMs into the design and saves each complete design/configuration as a checkpoint which is used for generating PBs. AMAH-Flex loads these checkpoint files separately and creates a full bitstream and a partial bitstream for each RM. This is the last phase and the design flow ends. AMAH-Flex leaves a note on the Vivado terminal at the end that states: *You need the following information (column address, row address, etc.) to calculate the FAR value that is essential for manipulating and relocating partial bitstreams.*

V. Evaluation

We evaluated various design flow stages to show the viability of our approach. In the *preparatory* phase, we tested whether we were able to collect all resources from different FPGA architectures. The tested devices were *Virtex 7 485t, Artix 7 200t, Kintex 7 325t and Zynq UltraScale+ 9eg.* As these resource files are device-specific and do not need to be recreated, they could in principle be delivered as libraries along with our tool. We were able to correctly determine the resources on all architectures so that working RPs could be instantiated on all architectures in the later process. The *floorplanning* phase successfully determined placements for all RPs on every tested architecture. The algorithm successfully worked with both, left-placed CPs and right-placed CPs. Our tool was able to make the resource and routing content of all RPs identical in the *make identical* phase. This shows, that the resource identification and floorplanning steps worked correctly for both, the 7-Series and UltraScale+ architectures. Full and Partial bitstream generation worked equally reliable. All generated partial bitstreams could be relocated by manipulating the FAR to suitable values targeting RPs within the same relocation group.

A. Mixed Design Sources

We test the usability of mixed sources on a design with two reconfigurable modules containing different FFT designs. The design itself contains various design sources: for the static design, a block design created in TCL containing the following components: Zynq US+ Core, System Reset, AXI GPIO, AXI Interconnect; for the reconfigurable design we used Verilog files describing a FFT block. The whole design used 1471 LUTs, 7 BRAMs, 9 DSP blocks, with 1300 LUTs, 4 BRAMs and 9 DSP blocks used by the reconfigurable FFT block. In the preparatory phase, the AMAH-Flex tool adds all design sources. The static module and the two RMs have been successfully synthesized and the necessary design reports generated.

B. Use-Cases

The designs contain four RPs and respectively two RMs and use the aforementioned FPGAs. In case of four RPs, the algorithm processes three RPs in 1-D, which differ only in Y axis and the fourth to the left of them, which differs in X and Y axis. This scenario indicates that the algorithm is successfully completed and can be used on the one hand for 2-D relocation and on the other hand for different FPGA boards. Moreover, this scenario shows that the algorithm can also be used to find and place more than two RPs. All scenarios were tested with two applications, first, one 16-bit multiplier and one 16-bit adder, and second, a shift-left register and a shift-right register. In both applications the RMs were relocated among the available RPs and their inputs and results recorded. Each RP, when it contained an RM, worked flawlessly and produced correct results according to its loaded module. Figure 5 shows the final placement result of the four RPs szenario on the Xilinx Zynq UltraScale+ FPGA. On the left side of the figure we can see there is no more feed-through routes using AMAH-Flex.

C. Results

Table I summarize the maximum number of reconfigurable regions within a relocation group based on 7 different benchmark designs with different sizes on different Xilinx FPGAs. From the table, we can see that the larger the region, the less the possibility of finding an identical one. Furthermore, the more

TABLE I: An overview of the placement results for maximum number of reconfigurable regions on both Xilinx 7-Series and UltraScale+ architecture.

Resource Utilization of Different Applications			Architecture				
			7-Series				UltraScale+
			Artix7 200t	Kintex7 325t	Virtex7 690t	Virtex7 485t	Zynq ZU9EG
CLB Tile	BRAM Tile	DSP Tile	Maximum number of reconfigurable regions within a relocation group				
200	0	0	19	28	100	107	43
[300-1200]	[0-20]	[0-40]	[8-3]	[27-7]	20	21	15
[1400-3500]	[0-30]	[0-140]	3	[7-6]	10	7	4

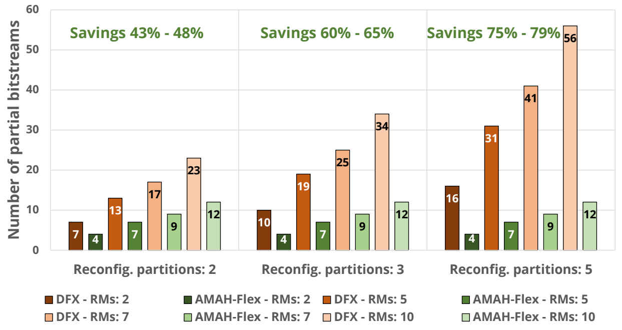


Fig. 6: An detailed overview of savings with AMAH-Flex compared to DFX.

heterogeneous the FPGA is, the less possibility there is to find several identical RPs. Nevertheless, in the 7-Series, AMAH-Flex finds mostly at least as many RPs as the number of vertical clock regions within an FPGA. In the UltraScale+ it is a little different since vertical clock regions also differ. Overall, we show that our design flow algorithm successfully completed and all phases of the design flow were executed successfully. Moreover, its design overhead and memory resource requirements are significantly better compared to the normal DFX: in the case of the previously mentioned use case with 2 RMs and 4 RPs for the same design, DFX would generate a total of 13 bitstreams, which can be computed using this equation:

$$P_{Bitstreams,\,DFX} = 1_{full} + M_{RP(blanking)} + (M_{RP} * N_{RM})_{partial} \quad (1)$$

In contrast, AMAH-Flex generates seven during design time:

$$P_{Bitstreams,\,Relocation:\,design\text{-}time} = 1_{full} + M_{RP(blanking)} + N_{RM,\,partial} \quad (2)$$

and requires only four during runtime:

$$P_{Bitstreams,\,Relocation:\,run\text{-}time} = 1_{full} + 1_{blanking} + N_{RM,\,partial} \quad (3)$$

Considering only the sums of the partial bitstreams (PBs) (including the blanking bitstreams, since they are only a special partial bitstream of the same size), there is a saving of 75%. This saving even increases with a higher number of RPs or RMs. Figure 6 illustrates the savings with AMAH-Flex compared to DFX in terms of max. generated PBs of a design with N RMs and M RPs.

VI. Conclusion

This work presented a new tool to create 2-D relocatable bitstreams. The design flow algorithm runs automatically and the user no longer needs to manually perform the *floorplanning* phase. Moreover, the design flow uses TCL scripts and was developed for the Xilinx Vivado Design Suite. It supports any kind of FPGA board from Xilinx starting with the 7-generation. AMAH-Flex can be used for 1-D and 2-D relocation as well as for designs with a large number of RPs. In this work, the RPs heights were one clock region. Placing one RP into the upper half and one into the lower half showed promising results, which will allow us to merge two half-heighted RPs into a single clock region for improved resource utilisation in a future extension of our approach. The AMAH-flex tool is published as an open source tool and is available at [16].

VII. Acknowledgement

This work is funded by the European Social Fund in Germany ESF and co-financed by tax funds based on the budget approved by the members of the Saxon State Parlament as part of the ReLearning project under grant agreement number 100382146.

References

[1] "Xilinx solutions," Xilinx. (), [Online]. Available: https://www.xilinx.com/products/silicon-devices.html (visited on 07/29/2021).

[2] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *20th Euromicro Conference on Real-Time Systems (ECRTS 2008)*, Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2008, pp. 147–155.

[3] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems," in *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005.

[4] C. Bolchini, A. Miele, and C. Sandionigi, "Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable FPGA systems," in *2011 21st International Conference on Field Programmable Logic and Applications*, Sep. 2011, pp. 532–538.

[5] C. Beckhoff, D. Koch, and J. Torreson, "Automatic floorplanning and interface synthesis of island style reconfigurable systems with GoAhead," in *Architecture of Computing Systems – ARCS 2013*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 303–316.

[6] A. Montone, M. D. Santambrogio, D. Sciuto, and S. O. Memik, "Placement and floorplanning in dynamically reconfigurable FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 4, pp. 1–34, Nov. 2010.

[7] 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*, Monterey California USA: ACM, Feb. 21, 2016, pp. 149–158.

[8] C. E. Neely, G. Brebner, and W. Shang, "ReShape: Towards a high-level approach to design and operation of modular reconfigurable systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 6, no. 1, pp. 1–23, May 2013.

[9] A. Lalevée, P.-H. Horrein, M. Arzel, M. Hübner, and S. Vaton, "AutoReloc: Automated design flow for bitstream relocation on xilinx FPGAs," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug. 2016, pp. 14–21.

[10] J. Rettkowski, K. Friesen, and D. Göhringer, "RePaBit: Automated generation of relocatable partial bitstreams for xilinx zynq FPGAs," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov. 2016, pp. 1–8.

[11] R. Oomen, T. Nguyen, A. Kumar, and H. Corporaal, "An automated technique to generate relocatable partial bitstreams for xilinx FPGAs," Sep. 2015, pp. 1–4.

[12] R. Zamacola, A. García Martínez, J. Mora, A. Otero, and E. de La Torre, "IMPRESS: Automated tool for the implementation of highly flexible partial reconfigurable systems with xilinx vivado," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec. 2018, pp. 1–8.

[13] Xilinx, "Vivado design suite tutorial: Dynamic function eXchange," p. 153, 2020.

[14] ——, *XAPP1335 - Isolation Design Flow for UltraScale+FPGAs and Zynq UltraScale+ MPSoCs*. Mar. 9, 2021, Version Number: v2.1.

[15] Xilinx Forum, *Isolated design flow (idf) and partial reconfiguration (pr) possible for zynq 7000?* Nov. 9, 2018.

[16] N. Charaf, C. Tietz, M. Raitza, A. Kumar, and D. Göhringer, *Amah-flex*, https://github.com/TUD-ADS, 2021.