HLS Enabled Partially Reconfigurable Module Implementation

DOI:
10.1007/978-3-319-77610-1

Document Version
Accepted author manuscript

HLS Enabled Partially Reconfigurable Module Implementation

Nicolae Bogdan Grigore, Charalampos Kritikakis and Dirk Koch

The University of Manchester
{nicolae.grigore, charalampos.kritikakis, dirk.koch} @manchester.ac.uk

Abstract. Making full use of the capabilities of the FPGA as an accelerator is difficult for non hardware experts, especially if partial reconfiguration is to be employed. One of the issues that arise is to physically implement modules into bounding boxes of minimum size for improving fragmentation cost and reconfiguration time. In this paper we present a method which automates the modules designing step, fulfilling module resource requirements and architectural FPGA constraints. We present a case study that shows how our automatic module implementation flow can be used to generate run-time reconfigurable bitstreams that are suited for stitching together processing pipelines directly from a Maxeler MaxJ HLS specification. This takes into consideration design alternatives, fragmentation, and routing failure mitigation strategies.

1 Introduction

HLS has made tremendous progress in recent years in improving design productivity of hardware systems. In particular for FPGA acceleration in datacenters, HLS and domain specific languages are commonly considered to be key technologies for succeeding with widespread FPGA deployment. However, one concern against this approach is the effect of *logic explosion* which expresses the situation that every line of (extra) HLS code translates somehow into extra logic on the FPGA and consequently in extra cost and more power. This situation gets crucial if major parts of the FPGA will remain idle for longer periods of time. For example: let us consider a driver assistance system with entire different object classifiers that were optimized for day and night modes. Then parts of the system (and correspondingly the FPGA) may not be used depending on the present mode. In this situation, partial reconfiguration at run-time is a viable option to optimize the module layout for optimizing resource utilization (e.g., by using reconfiguration to change between day and night object classification in our example). In general, whenever a system provides periods in time where functions are used mutual exclusively to each other, this is an opportunity for applying partial reconfiguration. This holds in particular if these periods are long enough in order to amortize the overhead induced for the reconfiguration.

The FPGA vendors Xilinx [1] and Altera [2] provide frameworks that allow developing run-time reconfigurable systems using HLS. In particular for the OpenCL language, industry reached a maturity level that allows software engineers and domain experts to build run-time reconfigurable systems without the need for extensive FPGA knowledge. This allows for non-FPGA experts to develop systems that can adapt to different requirements or workloads with the help of partial reconfiguration.

However, while this fundamentally is a strong achievement, present design methodologies and corresponding reconfigurable FPGA-based systems have important shortcomings that are not sufficiently addressed by the FPGA vendors. This includes in particular the flexibility in which partial reconfiguration can be used in a system. For example, present OpenCL frameworks support multiple reconfigurable regions that could host an accelerator module. However, a module is always only working at the position it was physically implemented and it is not possible to run a module implementation (given as a configuration bitstream)

at another position. Moreover, the physical partially reconfigurable module implementation is needed to be executed again whenever something changes in the static system (i.e. the part of the system providing I/O access to DDR memory etc.). Furthermore, the vendor flow does not foresee to use reconfigurable regions by multiple independently reconfigured and operated modules. Luckily, there are academic frameworks that allow the implementation of more flexible reconfigurable systems (e.g., OpenPR [3] and GoAhead [4]).

While such tools allow implementing reconfigurable systems with more capabilities, these tools are still designed to be used by FPGA experts. The goal of this paper is to provide a frontend for such tools (in this paper, we are building a frontend for GoAhead) that allows implementing partially reconfigurable modules directly from HLS descriptions by designers that do not need to be FPGA experts. In detail, this paper provides an automatic compilation framework for stream processing applications starting from HLS all the way down to a partial reconfiguration bitstream that supports flexible module placement, module relocation and multi module instantiation. We will provide a solution for compilation of MaxJ (Java) specifications to relocatable and stitchable stream processing modules (Section 5) in a dynamic dataflow system. We assume that an expert is providing a static system. For this, HLS compilers are used to retrieve module primitive requirements. With this, we will show how bounding boxes for modules can be automatically computed and implemented all the way to reconfigurable modules.

2 Related Work

As mentioned in the introduction, the major FPGA vendors are already providing solutions that allow building applications in HDLs for FPGAs that rudimentary use partial reconfiguration [1] [2]. Building partially reconfigurable systems introduces some extra level of complexity that commonly needs dealing with some low-level FPGA specific issues. In order to deal with such issues, design automation for partial reconfiguration has been researched.

With OpenPR [3] and GoAhead [4], tools have been developed that allow FPGA experts building reconfigurable systems with distinct features like module relocation and direct module to module communication. However, using these tools needs significant FPGA experience and a specific way of floorplanning for partitioning FPGA resources into static and run-time reconfigurable sections as well as for providing interfaces for integrating reconfigurable parts of a system. The work in [5] is focusing on automating the interface design using simulated annealing while in [6], [7], [8], and [9] the whole floorplanning process for the static system and/or partial modules was automated for RTL designs and demonstrated for a small number of modules. Static system only floorplanning was presented in [10] and [11]. The problem of physically designing relocatable modules was addressed, for example, in [12], [13], and [14]. There is a large body of rather theoretical related work (commonly without a system that is actually working on an FPGA) on automated floorplanning that is not listed here due to space limitations.

Physically implementing relocatable modules adds more constraints to be obeyed by design tools, and consequently, more potential points of failure for successfully completing the process all the way to the bistream level. While related work marks important automation steps, in this work we do not only provide a holistic solution to automatic floorplanning and interface synthesis for implementing relocatable modules, we in addition provide automatic mitigation strategies for the case that the physical FPGA implementation fails. This makes the whole backend flow that robust that it can be coupled with an HLS frontend such that reconfigurable modules can be implemented fully automated directly from high-level languages.

3 Model

Our goal is to build a design flow that can be used by non FPGA experts to take advantage of partial dynamic reconfiguration. For this, we assume that an expert must first design a

static system that defines a reconfigurable area to test the modules (representing the actual application). These modules will then be implemented by a non FPGA expert using HLS.

To do this, we must first define a model for the FPGA's reconfigurable resources and reconfigurable modules. All modern FPGAs from the vendor Xilinx contain a set number of resource slice types. These are usually SliceL, SliceM, BRAM and DSP (with some variations depending on the FPGA family). We can model the FPGA as a set of these resources or, in order to also express the exact sequential order of resource columns, as a resource string. This allows for modeling of the module placement process as a string matching problem.

Our automatic bounding box generation tool is generic in that it can work on any device as long as the following generic parameters are provided:

- Number of CLBs in a clock region
- Number of LUTs in a CLB
- Number of BRAMs in a clock region
- Number of DSPs in a clock region
- Total number of clock regions in the reconfigurable area
- Resource string of the reconfigurable area

This model fits directly to all Xilinx FPGA families including all 7-series devices.

Each module requires a number of resources in order to perform the task required, thus our initial representation has to provide at least the minimum requirements for each: number of BRAMs, LUTs and DSPs.

Using these two string representations for the FPGA and modules, as well as the number of primitives per resource column, we can find bounding boxes (as discussed later). Bounding boxes are represented in two complimentary ways: a set of three parameters specifying start position, width and height, as well as the resource string of the bounding box and number of clock regions required. Using string representations for the reconfigurable area and the reconfigurable modules allows for checking for feasible placement positions using simple string compare. The bounding box information can be used to build partially reconfigurable modules as shown in Figure 5. The example shows how MaxJ specifications are divided into a static part that is separated from the actual application (here the partial modules). Further details about this process will be given later in Section 5.

4 Bounding Box Generation

4.1 Overview

Our algorithm generates bounding boxes for a module based on the FPGA resource string modeling of the available resources in the reconfigurable region, the device specific primitive allocation to slices, and module primitive requirements.

During the generation phase we add more and more slices to the module string specification from the FPGA representation until all primitive requirements have been met. With this, we ensure to only implement modules for feasible module bounding boxes and that we identify all possible minimal design alternatives. Our system also takes into consideration multiple clock regions. The generated bounding boxes can span anywhere between one clock region and the entire height of the device (or reconfigurable region). This adds even further flexibility to the placement phase, making sure that the modules come with more possible placement positions, for allowing a much tighter overall packing.

In this work we assume scenarios that benefit from small module sizes as both examples will allow using multiple modules in a shared reconfigurable region. In both case studies we build modules to allow stitching together processing pipelines, while supporting direct communication.

4.2 Generation

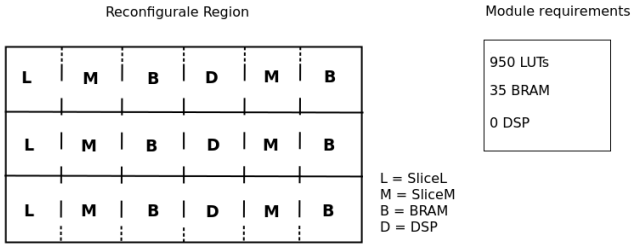

Fig. 1: Example of a reconfigurable region, spanning 3 clock regions, and some module requirements. The region is modeled with the shown alphabet, and a module is presented as a set of primitive requirements.

Let us assume that we need to find placement positions for the module in Figure 1 inside the shown reconfigurable region. Firstly, we must know the number of primitives provided in each column. These values are device specific and must be fed into our bounding box generator. In this example we will use a Zynq FPGA. On this device, we have the following number of resources for each column:

- SliceL : 40*8 LUTs
- SliceM : 40*8 LUTs
- BRAM : 20 RAM primitives (10 36 Kbit or 20 18 Kbit)
- DSP : 20 DSP primitives

The algorithm (illustrated in Figure 2) begins at the first available resource in the reconfigurable region. It checks if this resource contains primitives needed by the module. If so, it adds the slice to the module string and updates the module requirements to reflect that the primitive in the added slice have already been take into consideration. This step is repeated until all primitive requirements have been met, and the resulting module string represents a design alternative.

As stated before, in order to give the user as much choice as possible and to allow for fine grained and flexible module placement, our system looks for bounding boxes spanning from one to as many clock regions as the reconfigurable area has available. As such the steps above are repeated using incrementally more clock regions (i.e. increasing the height of the modules). Considering our example, we are looking at the bounding boxes starting with the first resource slice we can determine 3 placement positions as can be seen in Figure 3.

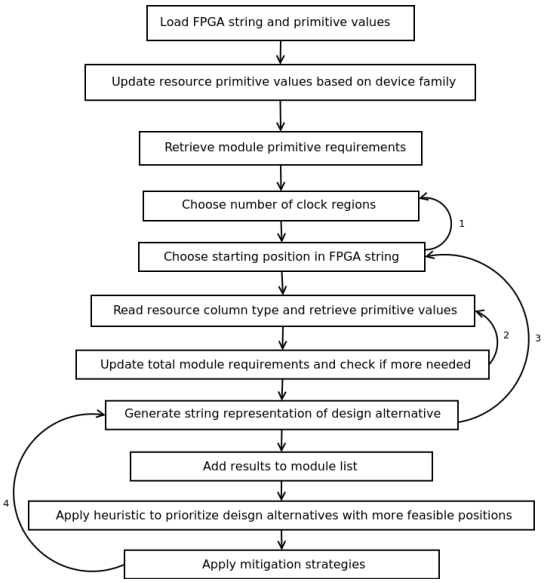

Fig. 2: Algorithm applied to each module specification. Transition 1 happens when all bounding boxes for the current number of clock regions were found. Transition 2 happens at every step until all resource requirements for the module are met. Transition 3 occurs once the bounding boxes starting at the current start position were found. Transition 4 allows for mitigation strategies to be employed if routing fails.

The bounding box generator exploits the fact that the smallest module (i.e. the module variant with the shortest resource string) that fulfills the resource requirements will result in the lowest internal fragmentation. As such, only the smallest design alternatives are considered at the end of this computation. This allows for the reduction of the run-time search space, whilst still providing high placement flexibility. For example, we will consider only two of the three module design alternatives for the first g position in the reconfigurable region provided:

- (LMBDMB) * 1 row
- (LMB) * 2 rows
- (LMB) * 3 rows (discarded as the two row variant has lower internal fragmentation)

All module bounding boxes generated will be continuous and rectangle. This means that unnecessary resources cannot be skipped. In our example, we see that the 1 row module generated contains a DSP resource slice even though DSPs were not necessary for the correct run of the module. Similarly the 3 row implementation uses more resources than the 2 row one, even though it only needs just as many. This means that there is a need for a step after the bounding box generation to determine which bounding boxes should be used for physical implementation.

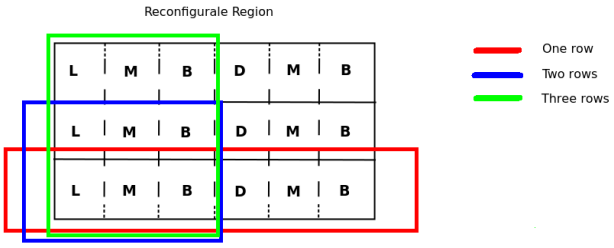

Fig. 3: Bounding boxes for the defined module in the reconfigurable region, starting only at the first resource slice.

In oder to further reduce the search space and still provide the user with flexibility, we employ a heuristic. Once the total number of design alternative is computed we sort the resulting list of bounding boxes in such a way that the alternatives with the most possible placement positions are at the start. Typically, a relatively small number of alternatives is sufficient to allow placement with little external fragmentation (i.e. unused resources between placed modules). This heuristic increases the chance that run-time placement results in better resource utilization.

One of the problems that can occur when creating bounding boxes is that, if they are defined aggressively small, there might not be enough resources left over for routing. Because our bounding boxes are rectangles and because we use resource columns as our placement atoms, the bounding box will likely leave some resources unused. Seeing as how routing a particular module can be difficult [15], the excess resources can improve the chance that routing will succeed and timing will be met.

As an extra precaution, we have implemented a method by which the module string can be updated to contain more resource columns as needed. Since routing requires the switch matrix only within a column, the extra resource (which we refer to as a slack variable) can be seen as a wild card (meaning any resource type can be used to ensure routing). This can be added before a placement method is applied. Finally, if timing still isn't met, we also allow for a fail message to be fed back to the generator in order to further increase the number of resources assigned for a module (i.e. one extra slice to the left and one to the right will most definitely solve the problem, but would be wasteful if not necessary).

Furthermore, our tool flow implements mitigation strategies that apply physical constraints that will be tried out to improve routability and performance (achieved clock frequency). This includes using switch matrices only at places of high possible congestion (e.g. the corner of the bounding box), as described in Section 5.4.

5 Case Study

This section focuses on applying an automatic partial module implementation flow on a Maxeler Max3 system using its dataflow model. The case study will include creating the static part in our design, as well as injecting a reconfigurable region amongst the automatically generated RTL code from an HLS tool. Moreover, we will focus on the extraction of HLS generated accelerators and the final mapping using the bounding box generator.

5.1 Maxeler system and Dataflow

Maxeler Workstations [16] are hybrid computing platforms that are using both a CPU and an FPGA to implement complex functions. The FPGA device is programmed by a Java dialect,

which is called MaxJ, in order to be more design friendly to non-FPGA experts, without having knowledge of HDLs.

The system uses an automatically generated interface infrastructure between the FPGAs and the rest of the system and, depending on the input interface, our system needs for example, PCI-e and/or memory. Moreover, Maxeler has a large userbase in academia and industry. Common applications domains include databases, medical applications, image/video processing, networking and so on [17].

We are currently using a Max3 Workstation, which provides a Virtex-6 XC6VSX475T FPGA from Xilinx, connected to the mother-board via PCI-e. The FPGA is surrounded by 24GB of DDR-3 memory and the host CPU is an Intel(R) Core(TM) i7-2600S CPU clocked at 2.80GHz CPU.

In order to use Maxeler, the designer has to focus on three basic parts, the CPU interface code, the main Kernel and the Manager. When all these three parts are developed, the MaxJ code is compiled to a corresponding RTL VHDL description, which is compiled by the ISE toolchain, until the final bitfile generation. The tool creates the Maxfile, which is a monolithic binary that contains the full static configuration of the FPGA and the host machine binary file. In order to run the system, Maxeler combines the CPU interface code and the Maxfile to run the computation and to retrieve the final result. Figure 4 is showing an overview of the whole design flow. It should be mentioned that Maxeler is offering a custom HDL interface, in order to allow the integration of hand-crafted RTL code to be used within Maxelers framework.

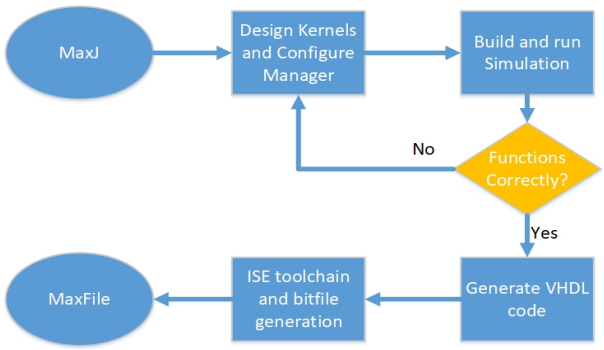

Fig. 4: Maxeler design flow from MaxJ to Maxfile.

5.2 Static System

Our system consists of a Maxeler interface that transfers data from the CPU to and from the FPGA, a custom HDL implementation or an HLS generated accelerator and a connection to the internal configuration access port (ICAP). The input to ICAP is needed to load the partial bitstreams of our partial modules. Inside the wrapper of our accelerator, we can replace the automatically generated accelerator with our reconfigurable region, which will host the partial modules. In order to split the accelerator from the static part, we can use Xilinx tools to set the accelerator as a top level entity and extract the netlist of the kernel. Then, we can use this netlist to have the full functionality of the kernel and create a partial module. We will focus more on the partial module creation in Section 5.3. The interface of the reconfigurable region is currently implemented entirely as a loopback device for a 512-bit wide datapath. In total, Maxeler's automatically generated modules around the accelerator, plus our reconfigurable region as an entity, are defined as the final fully static part of our system. Figure 5 shows a detailed overview of the steps we follow to generate the full static system of the dynamic part and the reconfigurable region.

For integrating a reconfigurable region into the design, we manually floorplanned the static system, by taking into consideration where Maxeler maps the rest of its system. That includes the placement of I/O cells for the PCI-e and DDR3 memory connections and the surrounding modules. Given these constraints, the Maxeler implementation is not using the corners of the device. Hence, our reconfigurable region is constrained to be placed in the upper right corner of the FPGA, which is not used by the surrounding Maxeler system.

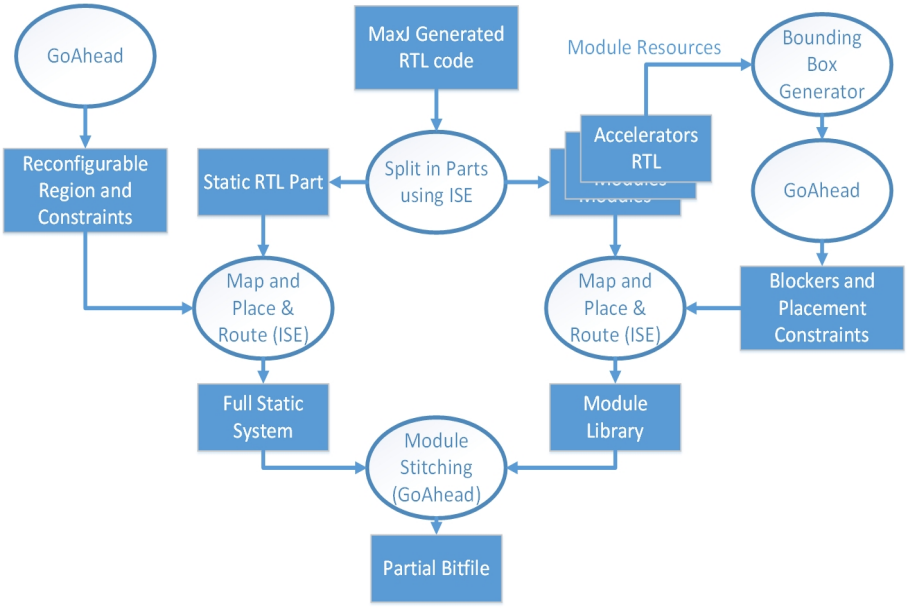

Fig. 5: Proposed design approach: MaxJ generated system split into a static system with reconfigurable area and a module library.

The reconfigurable region is entirely generated by GoAhead and placed and routed with ISE tools. The final reconfigurable region is depicted in Figure 6a. To create the reconfigurable region, we define a bounding box in GoAhead. Then, inside the reconfigurable region, we place a series of registers in vertical fashion on the left side, middle and right side of our region, that help us route in a specific way across our defined area. Additionally, placement and routing constraints are generated that prevent the Xilinx place and route tool to use any resources within the reconfigurable region.

5.3 Implemented Modules

For the whole process we follow a standard pattern which is depicted in the right side of Figure 5. For each one of our modules, we programmed our kernel in MaxJ and went through the Maxeler compilation flow, until the HDL code is generated. Then, we automatically analyze the generated HDL code, in order to identify and separate the kernel hierarchy from the surrounding Maxeler system (Figure 5). More specifically, we set the kernel as a top-level module and compile the entire kernel hierarchy into one netlist file. This will also report resource requirements to the automatic bounding box generator, which will in turn return the starting point, the width and a resource string of the module. For each module we get a

different resource string or a set of resource strings, that refer to different positions, depending on the requirements of the implemented module.

Subsequently, we use GoAhead to generate placement constraints and to create a blocker around the module, such that the module routing will never cross the module borders. With the generated constraints we can use the Xilinx toolchain to fully map and route the module. One of our main challenges is to route through and back of our module, precisely as we did in our reconfigurable region. In order to ensure this routing, we place a vertical series of registers before and after the module, that we call connection macros, that act as the interface of our partial module. It should be mentioned that the clock signals to ensure the routing will exactly match the routing used in the reconfigurable region.

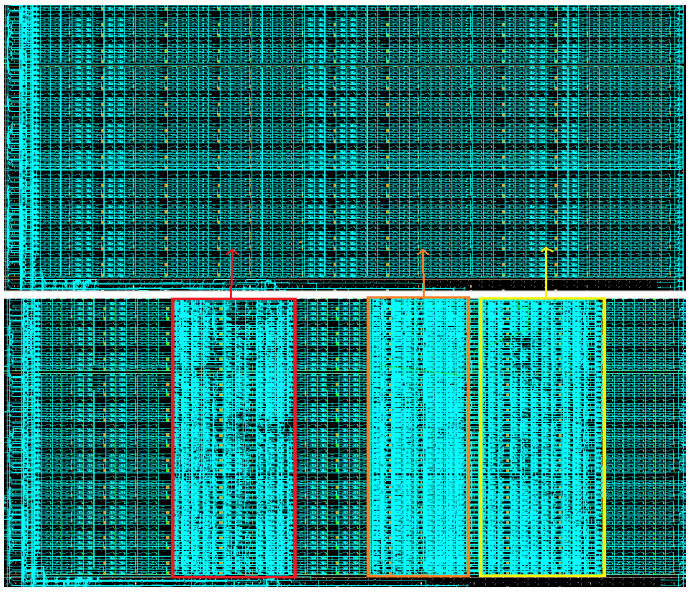

Fig. 6: (top) Empty reconfigurable region. (bottom) Placed partial modules in reconfigurable region.

Finally, we can cut off only the module and save it in a netlist file format in a module library. Again, the cutting of the module from the final routed file is done by GoAhead. This tool allows to select a specific area, by given constraints like height and width and a starting point, in order to extract only the module. This module can now be placed into the reconfigurable region of the static system. We use string matching to find module placement positions, following the module presented in Section 4. This placement allows us to stitch together different stream processing modules (that are generated by MaxJ). After each stitching process, we generate a partial bitstream for the design. In order to place the module in the Maxeler system, we need to have an input in our device, that targets the ICAP port. More specifically, we can generate a partial bitfile, containing the position and the configuration of a partial module. This bitfile can then be used for reconfiguration through the ICAP port.

Our current module library consists of 6 image processing functions. Those are a Sobel filter, brightness correction, a gaussian filter, an RGB to greyscale filter, a skin color detection and a mean filter. All of those functions are generated entirely by Maxeler and MaxJ code.

It is possible to place more than one reconfigurable module in the reconfigurable region. An example of fully placed modules is shown in Figure 6. More specifically, Figure 6a)

illustrates an empty state of our reconfigurable region. using the resource string model of this region, the placer can calculate positions for each module. Figure 6b) shows the final placing of 3 distinct modules in the same reconfigurable region. In that way, we can stitch a parallel pipeline of modules, that can take an input stream and apply different functions on it.

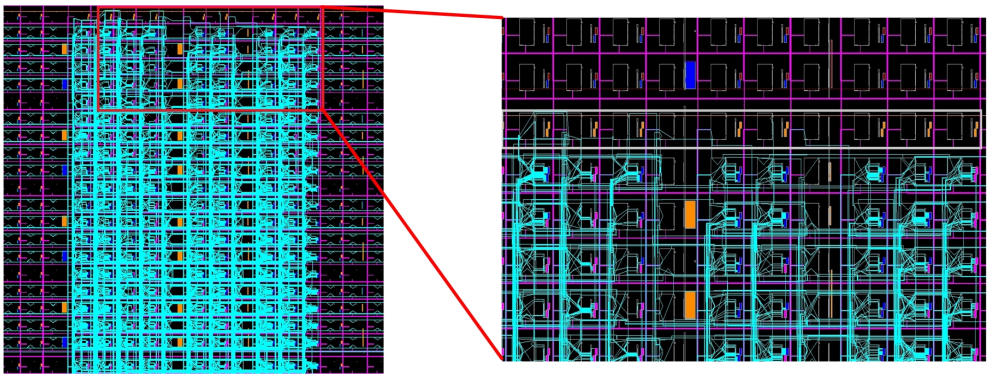

Fig. 7: Mitigation Strategy 2: We leave the top and bottom of the partial module unmapped and we just use the switch matrices.

At this point, we should point out that some parts of the modules are underutilized. The restricting factor is practically the internal design of the Virtex 6 FPGAs. To be more precise, if a module needs more than 4 rows of CLBs, then, by definition, the design will have a row of either BRAMs or DSPs.This will result in internal and external fragmentation, in order to fulfill the CLB resource requirement. An example is shown in Figure 6b, where the first module (red boxed module) has two columns in the middle which are a row of DSPs and a row of BRAMs. However, we can observe that the rest of the rows, which consist of CLBs, are almost fully populated, thus a smaller bounding box could not be chosen.

5.4 Mitigation Strategies

As an additional function, our physical implementation flow is able to handle failures during mapping or routing. For example, if a module does not get mapped, we will extend the bounding box left or right by an additional column. In the case of a routing failure, the tool can relax the routing inside the bounding box, using 3 different strategies, that are tried out in the following consecutive order.

- The tool can block the placement in the corners of the bounding box, because the design tends to be heavily congested in the corners. So we are leaving them unused but, by taking advantage of their routing resources (switch matrices), we provide locally a higher ratio of routing logic.
- We are leaving the top and bottom side unmapped, as it can be seen in Figure 7. Figure 7 left shows the fully routed module, while on the right side of Figure 7, the unmapped CLBs on the top row are depicted, inside the gray box. This solution can solve a rather small unroutable situation, without spreading the design in more rows.
- The last, most efficient but also expensive strategy, occurs by using an additional layer as a frame around the module, in which we will only take advantage of the routing resources. Figure 8 depicts that kind of situation. In this case, the mapping will be done entirely in the inner side of the frame, while we will use only the routing resources of the frame. In the left side of Figure 8, the full partial module is presented, while on the right, a

zoomed in representation of the partial module is presented. The above solutions can offer significant design alternatives for our modules, without the risk of having external fragmentation.

Implementing modules in bounding boxes includes more constrains on the physical implementation. However, in some cases, routing is not possible, for a module with the given constraints. As an example, in the four corners of the bounding box, the router has only available about half of the available routing wires. This situation may result in an unroutable situation and that is why mitigation strategies are vital for such kind of implementations.

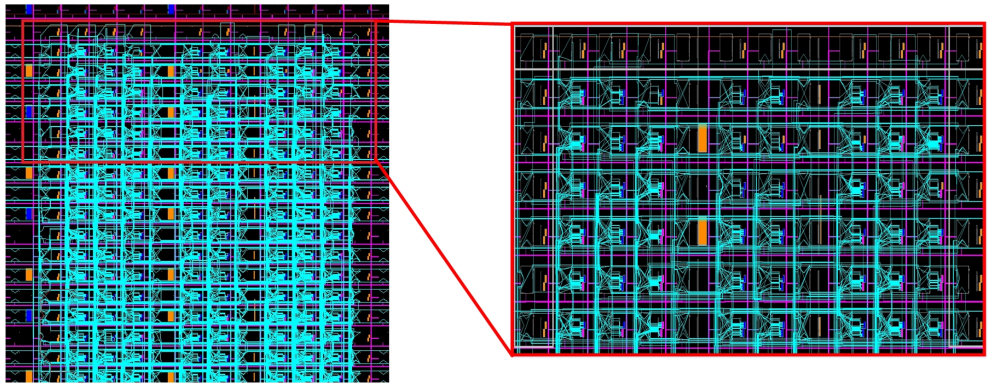

Fig. 8: Mitigation Strategy 3: Unplaced, but routed frame around the module, to relax routing if necessery.

6 Conclusion

In this work we presented a tool flow that automates the generation of partially reconfigurable stream processing accelerator modules directly from HLS using minimum feasible module bounding boxes. We have discussed in detail how our bounding box generator incorporates the heterogeneous resource layout of FPGAs using a string model for reconfigurable regions as well as for the reconfigurable modules. We are able to generate physical implementation alternatives for enhancing module packing at run-time. Furthermore we incorporated automatic mitigation strategies to get even highly congested modules physically implemented.

We demonstrated our approach using a case study that centers on a Maxeler Max-3 Dataflow processing system. Our flow allows it to automatically generate relocatable partial bitstreams directly from MaxJ. We also showed how these bitstreams can be used at run-time. The here presented methodology is quite universal and can be applied to other (Xilinx) FPGAs and other HLS tools (e.g.,Vivado HLS).

With the proposed flow we allow for non FPGA experts to make better use of FPGAs including powerful tools such as dynamic partial reconfiguration. This provides also means to close a semantic gap that is that commonly a few functions out of a larger library are called dynamically in software and the here presented tool provides an important piece to translate this approach into FPGA acceleration.

Acknowledgements

This work is kindly supported by the European Commission under the H2020 Programme with the project ECOSCALE (grant agreement 671632) and with the project Reconfigurable

Tera Stream Computing, funded by the Defence Science and Technology Laboratory under grant DSTLX10000092266.

References

1. L. Wirbel, "Xilinx SDAccel," 2014.
2. J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.
3. A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An open-source partial-reconfiguration toolkit for Xilinx FPGAs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on.* IEEE, 2011, pp. 228–235.
4. "GoAhead Project," http://www.mn.uio.no/ifi/english/research/projects/cosrecos/goahead/, 2017.
5. J. M. Carver, R. N. Pittman, and A. Forin, "Automatic bus macro placement for partially reconfigurable FPGA designs," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays.* ACM, 2009, pp. 269–272.
6. K. Vipin and S. A. Fahmy, "Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq," in *Adaptive Hardware and Systems (AHS), 2015 NASA/ESA Conference on.* IEEE, 2015, pp. 1–8.
7. C. Beckhoff, D. Koch, and J. Tørresen, "Automatic floorplanning and interface synthesis of island style reconfigurable systems with GoAhead." in *ARCS.* Springer, 2013, pp. 303–316.
8. C. Beckhoff, D. Koch, and J. Torresen, "GoAhead: A partial reconfiguration framework," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on.* IEEE, 2012, pp. 37–44.
9. M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio, "Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 151–164, 2017.
10. F. Mao, Y.-C. Chen, W. Zhang, H. H. Li, and B. He, "Library-based placement and routing in FPGAs with support of partial reconfiguration," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 4, p. 71, 2016.
11. A. Otero, A. Morales-Cas, J. Portilla, E. de la Torre, and T. Riesgo, "A modular peripheral to support self-reconfiguration in SoCs," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on.* IEEE, 2010, pp. 88–95.
12. A. Lalevee, P.-H. Horrein, M. Arzel, M. Hübner, and S. Vaton, "AutoReloc: Automated design flow for bitstream relocation on Xilinx FPGAs," in *Digital System Design (DSD), 2016 Euromicro Conference on.* IEEE, 2016, pp. 14–21.
13. F. Ferrandi, M. Novati, M. Morandi, M. D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead," in *System-on-Chip, 2006. International Symposium on.* IEEE, 2006, pp. 1–4.
14. 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.* IEEE, 2005, pp. 8–pp.
15. A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays.* ACM, 1999, pp. 69–78.
16. M. Technologies, "Multiscale dataflow programing," 2014.
17. "Maxeler App Gallery," http://appgallery.maxeler.com/#/, 2017.