Journal of Signal Processing Systems (2020) 92:887–905
https://doi.org/10.1007/s11265-020-01545-y

Enabling Dynamic System Integration on Maxeler HLS Platforms

Charalampos Kritikakis[1] · **Dirk Koch**[1]

Received: 29 November 2019 / Revised: 11 March 2020 / Accepted: 5 May 2020 / Published online: 9 August 2020

Abstract
High Level Synthesis (HLS) tools enable application domain experts to implement applications and algorithms on FPGAs. The majority of present FPGA applications is following a stream processing model which is almost entirely implemented statically and not exploiting the benefits enabled by partial reconfiguration. In this paper, we propose a generic approach for implementing and using partial reconfiguration through an HLS design flow for Maxeler platforms. Our flow extracts HLS generated HDL code from the Maxeler compilation process in order to implement a static FPGA infrastructure as well as run-time reconfigurable stream processing modules. As a distinct feature, our infrastructure can accommodate multiple partial modules in a pipeline daisy-chained manner, which aligns directly to Maxeler's dataflow programming paradigm. The benefits of the proposed flow are demonstrated by a case study of a dynamically reconfigurable video processing pipeline delivering 6.4GB/s throughput.

Keywords Partial reconfiguration · HLS · Stream processing · Image processing · Maxeler · FPGAs

1 Introduction

Over the last decades, data processing applications were getting constantly more complex. Scaling the number of cores alone is no longer a feasible solution any more due to the increasing utility costs and power consumption constraints [1]. Thus, there is a growing need for delivering more computing capability to process larger quantities of data. Moreover, due to the fact that data volumes are often growing substantially faster than processing capabilities, data centers need to increase remarkably to catch up with processing requirements.

Heterogeneous computing systems are considered to be a viable solution for performance increase of computing systems. The intent in using FPGA technology is stimulated by the fact that FPGAs offer significant speedup and energy efficiency [2]. For example, FPGAs were used as accelerators in the Baidu Search Engine [3], demonstrating significant throughput and throughput-to-energy efficiency ratio (of over 20x). These achievements resulted in industry adopting FPGA acceleration widely and all major cloud service providers have corresponding offerings (e.g. Amazon F1 instances [4], Alibaba cloud services [5] and Microsoft Azure [6]).

Over the last years, the use of High-Level- Synthesis (HLS) tools, which create FPGA designs from high level languages such as C, C++ or Java instead of Hardware Design Languages (HDL), is on the rise. Now, the FPGA vendors Xilinx [7] and Altera [8] provide HLS frameworks that allow developing run-time reconfigurable systemsusing HLS. Although,allowing non FPGA experts to implement partially reconfigurable designs is an achievement on its own, there are still some limitations in the capabilities of corresponding reconfigurable systems. Major limitations are that reconfigurable modules cannot be relocated on the FPGA and that the whole system (static and dynamic part) needs to be recompiled, should a change takes place in either part. Additionally, each reconfigurable region cannot host multiple individual reconfigurable modules at the same time and there is no support to directly communicate between reconfigurable modules. This implies that processing pipelines cannot be

✉ Charalampos Kritikakis
charalampos.kritikakis@manchester.ac.uk

Dirk Koch
dirk.koch@manchester.ac.uk

[1] The University of Manchester, Manchster, UK

0 implemented by directly stitching reconfigurable modules together, when using the vendor flow. However, there are academic frameworks that allow the implementation of more flexible reconfigurable systems (e.g., OpenPR [9] and GoAhead [10]).

In addition to providing HLS enabled system flexibility through partial reconfiguration, one characteristic of our systems is that DDR memory controllers are kept active during reconfiguration. This allows for keeping data stored in DDR memory intact even if the FPGA gets reconfigured. This is an improvement over existing Maxeler systems where all the data stored in DDR memory gets corrupted after configuration. Maxeler systems offer large capacity FPGAs with access to a large amount of DDR memory (tens of GBs). Maxeler offers stand alone systems with accelerator cards for PCI-e as well as network offload compute engines. Currently, a Maxeler system that is reconfigured but required to keep the local DDR memory contents intact would have to save the memory contents temporarily to a host machine which can cost tens of seconds due to the large amount of DDR memory and the relatively slow PCI-e speed available.

Moreover, our work addresses the portability of implemented modules. This means that we allow physically implemented accelerators to be ported to another application. In our approach, multiple accelerator modules form a library and by using predefined physical interfaces, we allow integrating accelerators directly from a netlist or at bitstream level.

In summary, in this paper we are presenting a system that allows *the dynamic integration of stream processing operators* based on an *automatic HLS approach*. This allows Maxeler users to create dynamic system implementations that are independent amongst the static system and the accelerators and that can be arbitrarily stitched and daisy-chained in a reconfigurable region of a project. The main contributions of this work include:

- A Maxeler language extension and its corresponding automatic front-end preprocessing mechanisms to enable dynamic Maxeler Java implementations directly from an HLS application (Section 4).
- An automatic mid (Section 5) and back-end (Section 6) toolflow for run-time reconfigurable systems that enables application domain experts to build FPGA partial configuration bitstreams from HLS code that is written in Maxeler's Java dialect.
- Enabling design choices through software (Section 4.5), as well as portability, module replication, and module stitching on Maxeler systems (Section 6.2).
- An implementation and evaluation of a dynamic video processing system running on a MAX3 platform (Section 7).

2 Related Work

The first work applying partial reconfiguration on a Maxeler platform was the MSc project of J.J. Jensen [11]. In that work, the author presents a library of database accelerators that can be dynamically modified at run-time. The goal of that project was to show that some database queries can be accelerated in hardware by stitching together basic building blocks at run-time. The building blocks would be chosen depending on the query that is currently executed. The project was not fully implemented on an FPGA. Although, the author proves that accelerating queries in FPGAs dynamically can offer significant speedup by putting together smaller building blocks in order to create a datapath. In that project [11], all reconfigurable modules had been implemented as RTL designs rather than using MaxJ specifications.

R. Cattaneo in [12] also focuses on dynamic reconfiguration on Maxeler. That work considers a video streaming application like the one considered in this paper. The application includes 4 filters, which are Noise cancellation, Greyscale, Edge detection, and Threshold filter. The achievable throughput is 64.8 MB/s on a single stream and [12] also considers multiple regions and multiple stream operators with 4 input streams and 8 regions, which resulted in an aggregated throughput of 176.4 MB/s. Our proposed system in this paper can achieve a more than 36x faster throughput than the there presented single stream application on the same Max3 platform (see Sections 7).

The work in [12] does not include relocation and replication of the generated accelerators. However that work does suggest the chaining modules but at compile time. This would be prohibitively slow for run-time adaptive systems such as database acceleration where the exact chain of the modules is only known at run-time. In addition, [12] does not consider automatic placement, which is essential when introducing a low-level implementation for non-FPGA experts.

3 Maxeler Dataflow Programming Model

This section describes the current Maxeler tool, in terms of the user experience, the design methodology, the programming model, and the Maxeler compiler. Additionally, we will introduce our proposed dynamic flow in a top-down view, as will be described in the next sections.

3.1 Maxeler Static Dataflow

Maxeler Workstations are hybrid computing platforms that use CPU and FPGAs. The FPGA devices provided by Maxeler are programmed by a Java dialect, which is called

MaxJ. The Maxeler design flow saves a significant time for debugging, as it allows verifying the functional correctness through development tools such as MaxDebug on the CPU. Also, the system has an automatically generated interface between the FPGAs and the host and, depending on the input interface, a system may use PCI-E and/or Memory. Once an application is debugged, the MaxJ code will be compiled into a bitstream for the FPGA using a compile by correction approach. As in most of the HLS tools, a Maxeler user is not required to have FPGA specific knowledge to use the platform. However, for certain FPGA specific optimizations offered by Maxeler, basic knowledge of the devices is required (e.g. pipelining depth, memory configuration and stream clock frequency).

Maxeler systems are used in industry and academic projects and had shown outstanding performance, for example for geological applications [13], biophysical modeling [14], and convolutional neural networks [15]. Maxeler provides a large library of accelerators for applications of different domains, such as Databases, Medical Application, Image/Video processing, Networking, and so on [16].

In order to implement an application on a Maxeler system, a designer has to provide three basic parts 1) the *CPU interface code*, which is C code that handles the data that flows between the software on the CPU and the accelerator running on the FPGA, 2) the *Kernel/Kernels*, which contain the functionality that will be implemented on the FPGA, and 3) the *Manager*, which combines the accelerators functionality (i.e. kernels) and handles the on-board (between kernels) and off-board data movement (between the host and the FPGA). The functionality of the kernels and the manager is entirely described through Java and MaxJ functions [17]. After debugging and simulating the MaxJ code, Maxeler compiles all the existing kernels and the manager of the system through the *Maxeler MaxCompiler*. The MaxCompiler analyzes the MaxJ code and generates the corresponding VHDL code. After the VHDL code generation, the Xilinx FPGA vendor compilation tools perform the implementation, which includes synthesis, technology mapping, place and route, and, finally, generating the bitfile. In this paper, we are using a Maxeler system with a Xilinx Virtex 6 FPGA, which implies using the ISE tool. After the vendor compilation process, the MaxFile is generated. The MaxFile is a monolithic binary which contains both the full static configuration of the FPGA and the host machine binary file. In order to run the application, Maxeler uses both the CPU interface code and the MaxFile, which is executed on the FPGA. Figure 1 illustrates the whole design flow.

The manager orchestrates the connections between the software and the hardware and within the hardware application itself for each Maxeler project. Each Maxeler project has exactly one active manager file which contains the full implementation of the desired application. The generated files before and after the RTL generated version of the manager file are handled by the MaxCompiler. The manager file also contains the *software interface commands*. Each manager can have multiple run interfaces, which are enabled for more complex systems, where specific parts of an implementation may be enabled. Moreover, a user can define these interfaces in the manager for controlling the whole application. The available Maxeler interfaces are *Basic static, Advanced static, and Advanced dynamic*. Basic Static allows a single function call to run the application, while the Advanced Static interface

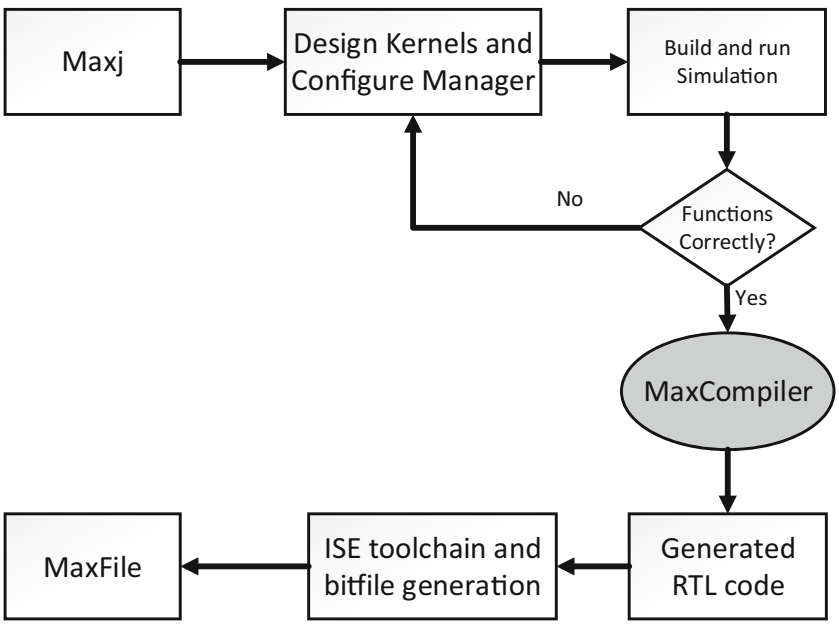

Figure 1 High level view on the Maxeler static MaxJ to MaxFile flow.

allows control of loading the applications and setting multiple complex actions. Advanced Dynamic allows for full scope of dataflow optimizations, fine-grained control of allocation, and de-allocation of all dataflow resources.

In addition to the implementation of applications in MaxJ, Maxeler offers a Custom HDL interface to allow the integration of RTL code to be used within a Maxeler system. Using this flow, a designer provides VHDL or netlist files, while the Kernel acts as a wrapper for the connections with the Manager. The final result is still a MaxFile, but the compiler keeps the RTL files and generates the logic before and after the added RTL hierarchy.

Maxeler allows for parallelizing applications across multiple FPGAs by using MaxRing. MaxRing provides a streaming connection between FPGAs for allowing the composition of long multi-FPGA processing pipelines.

3.2 Dynamic Maxeler Dataflow

This work aims at introducing dynamic partial reconfiguration on Maxeler platforms. To achieve this inside the given toolflow, there must be additional processes that allow the current tool flow for implementing static only systems to generate the code that comprises the parts of the *dynamic system*. This results in two parts: 1) the static implementation and 2) the dynamic accelerators/kernels. As a high level summary, we implemented 3 different levels of background processing, the front, mid, and back-end.

Figure 2 depicts the proposed approach to allow the current static only framework of Maxeler to generate dynamic systems. This figure also correlates with Fig. 1 and illustrates where at each level we have modified the static Maxeler flow. In Fig. 2, the flow is not changed until after the programing of the kernels and the manager in MaxJ. Then, the front-end process modifies and separates the static implementation from the accelerators before progressing to the MaxCompiler to generate the respective RTL code.

After the RTL code generation, additional changes occur in order to allow the implementation of independent building blocks. This process is handled by our mid-end process that will modify the generated code (as detailed in Section 5) accordingly and make this modified code available for physical implementation. The physical implementation is handled by the back-end process, which uses a combination of academic and Xilinx tools to generate the final physical implementation of the toolflow.

In contrast to Fig. 1, our flow generates two outputs. This includes a MaxFile of the project (i.e. a monolithic binary consisting of the software application binary and the configuration bitstream), that will be generated by the static implementation of the dynamic project. However, in every dynamic application on FPGAs there are a number of dynamic accelerators to switch at run-time. Those accelerators will be implemented as partial modules that will constitute a partial module library.

4 Dynamic MaxJ Specifications

Introducing a flow for partial reconfiguration in an HLS tool requires that the HLS language allows for describing the implemented system subparts (i.e. dynamic accelerators, static accelerators, and static system). Currently, a Maxeler user can not describe a dynamic system through MaxJ. In order to provide such functionality for this HLS language, it was decided to introduce a *language extension* for MaxJ.

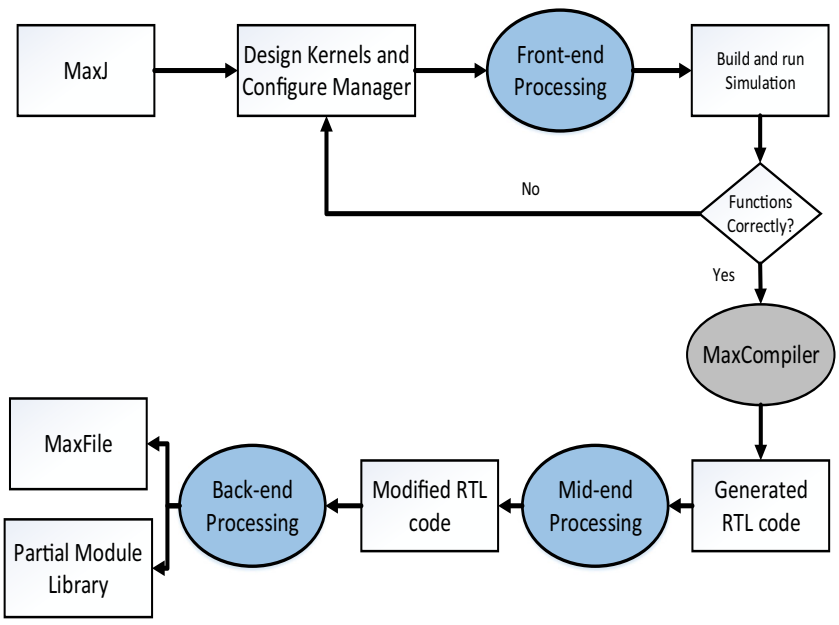

Figure 2 Proposed Maxeler dynamic MaxJ to MaxFile flow. The blue oval functions correspond to the added subprocesses necessary for the dynamic project implementation.

By introducing an extended version of the language, a preprocessor is necessary for processing the predefined extensions accordingly.

This section analyzes the existing limitations of MaxJ for supporting partial reconfiguration. Those limitations are tackled through a language extension. This section also introduces a preprocessor to parse the extended version of the language. In addition, we will analyze how the preprocessor and the language extension was integrated with the MaxCompiler to constitute a dynamic front-end flow. Finally, the here presented flow allows additional out-of-the-box design choices that enable reusability of the generated components.

4.1 Existing Limitations

The first goal while designing a language extensions is to allow all previously existing functions to be available in the extended version of the tool without any changes. With this, we allow all types of connections from the software to the FPGAs and vice versa, for allowing design simulation and the tool's debug/error messages to function with the extended dynamic applications, as used by Maxeler. With this, we maintain the high-level user experience as know from Maxeler.

For implementing a module library of reconfigurable accelerator modules, the Maxeler compiler can not extract RTL generated code of kernels that is not going to be used in an application (i.e. code that is not instantiated in the manager file). In other words, there is no way for the user to force the tool to generate the dynamic components (i.e. dynamic kernels) unless we instantiate them in the Maxeler project. However, instantiating them is not an option as the tool will generate additional connections for those kernels that can not be easily removed by an after-processing mechanism.

Another issue is that the MaxCompiler strictly compiles only one manager file (see Section 3.1). This makes sense for static FPGA systems. However, for run-time reconfigurable systems that introduce not only kernels as partial modules, but also multi-kernel blocks, a user will need a new HLS description mechanism. Therefore, we introduced multiple manager-like functions that allow the instantiation of an arbitrary number of kernels and arbitrary connections amongst those kernels.

Lastly, we have to provide a way to allow the definition and the creation of dynamic connections in the Maxeler manager. In the original Maxeler approach, a user could set different interfaces to be orchestrated from software, as mentioned in the previous section. In a Maxeler design, a user can operate on multiple streams. However, some streams and some parts of the design may be independent from the rest of the system, while not being mutually exclusive. An example is shown in Fig. 3, where the implementation has an *ICAP* module and *static kernel 1* that are independent from the reconfigurable pipeline. The user can choose to load data only to the ICAP module, without triggering the other 3 streams or just perform a computation on the static kernel 2. Note that this problem is different than partial reconfiguration, as the three depicted streams in the figure are not mutually exclusive to each other. Due to the dynamics available through run-time reconfiguration, we need to be able to set static and dynamic connections from the manager file itself.

4.2 Language Extension

In order to extend the MaxJ dialect with the concept of run-time reconfiguration, we introduced the concept of *PRGroups*. A PRGroup is a manager-like Java function that can contain from one to an arbitrary number of kernels. In addition, a PRGroup is described as part of a Java function, that can contain the same capabilities as a manager constructor in the original Maxeler model. A PRGroup can have arbitrary internal connections between the instantiated kernels by the user. The PRGroups allow a user to define building blocks containing any number of kernels and also switching of accelerator modules/kernels in a dynamic application.

A reconfigurable system may cluster a couple of reconfigurable modules to one atomic unit. We foresee this by the introduction of *PRGgroups* which are containers for one or partially reconfigurable modules. PRGroups are instantiated in the *PRManager* which can contain an arbitrary number of PRGroups as well as connections to and from the static part of the design. Alongside the PRGroups, the PRManager contains the static kernels that constitute the static part of the dynamic system.

The static instances and the PRGroups constitute a dynamic design. The PRManager contains a description of the dynamic design as shown in Fig. 3. The static system is composed of the peripheral modules (e.g. for DDR memory or PCI-e), static accelerators, an Internal Configuration Access Port (ICAP) to reconfigure the device, and the reconfigurable region that will host the partial modules.

Currently, in the original Maxeler programming approach, a user needs to set the connections between the CPU and the FPGA. However in a dynamic system, the static part and the dynamic accelerators may have different communication interfaces. Those communication interfaces correspond to the streams connected to the static part of the system. The dynamic streams will specify the connections between the reconfigurable region and the dynamic accelerators. Therefore, the PRManager has to be able to define and distinguish between static and dynamic connections. This was done by introducing a different type of interface

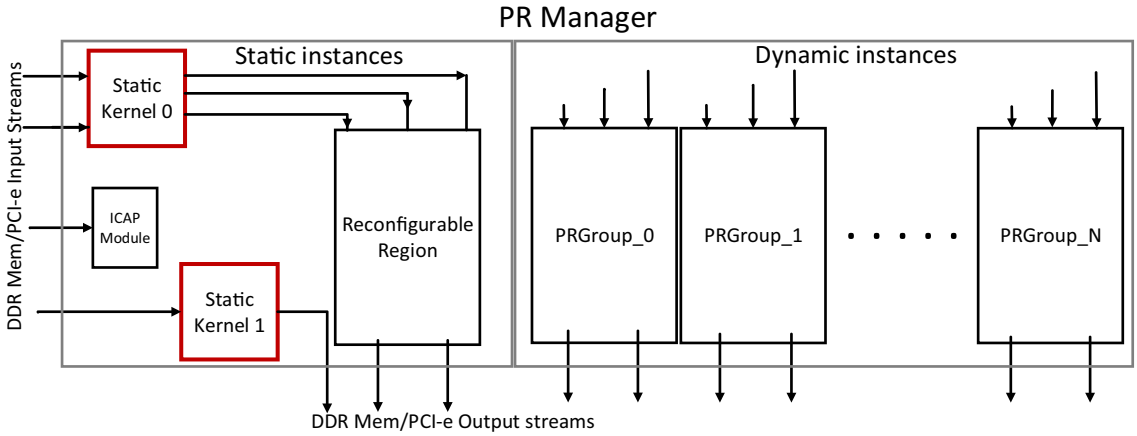

Figure 3 Block diagram describing a PRManager. The PRManager is split in 2 parts: the static instances and the dynamic instances. A PRManager can contain arbitrary connections and an arbitrary number of static kernels (with at least a Reconfigurable region module and an ICAP module). The static instances of the PRManager can also contain full static connections like the one that is connecting static kernel 1, where its input and output do not interact with the dynamic system, while a single stream needs to be connected to the ICAP module. In addition, the PRManager can contain multiple PRGroups as part of the dynamic instances. PRGroups have to fit the reconfigurable region and the interface. The interface needs to match all PRGroups and the reconfigurable region instantiation.

function, which is *setDynamicStream* (as an extension of setStream in the original Maxeler approach) to define the interface of the region that will host the partial modules, as well as the partial modules themselves. An example of a corresponding PRManager is shown in Fig. 3 where the reconfigurable region and the PRGroups have the same interface. This I/O interface is set by the function setDynamicStream, while with setStream, a user can define the interface of the static system. This includes all the inputs and outputs from and to the PCI-e and DDR memory. In Fig. 3, the output streams of the reconfigurable region need to be instantiated both by setDynamicStream and setStream.

The PRManager must contain at least 2 static instances: the *reconfigurable region* and the *ICAP core*. The reconfigurable region is given through our Maxeler extension library and it is implemented as an empty module with a connection from the input to the output. The reason for this is that we need a way to force the tool to generate the necessary physical connections for the region. The ICAP core is necessary to load the partial module configuration bitstream into the FPGA. In the here presented system, there may exist accelerator kernels that will not be swapped at run-time, as illustrated in Fig. 3. Those accelerators are called static kernels and they will be part of the static system. Static kernels are useful for parallel processing and pre or post-processing for the partial modules. Static kernels are kernels that are not specified as PRGroups.

A code example is depicted in Listing 1. The code includes the definition of a PRManager as it is done normally by Maxeler programmers. Then, the implementations for the ICAP and the reconfigurable region are included in line 3 and 6, alongside their connections in lines 5 and 9–13. The PRGroups definition occurs in line 15 and it contains 2 kernels that are instantiated in line 16 and 17 and similarly their connections in lines 19–23. Whatever is out of a PRGroup function (e.g. line 14–23) but within the PRManager class (e.g. line 1–24), it will be considered as a static accelerator, as it is for the Partial region in line 6 and the icapModule in line 4. As it is essential in the programming model of Maxeler, the interface code that handles the connections between software and hardware should be created as shown in line 27. however, currently we have another instance within the FPGA design which is the reconfigurable region. Thus, the connections to the region need to be declared at this point. First the static connections are declared in lines 37–39 and then, the connections to the partial region are set as shown in lines 41 and 42 with the introduced *setDynamicStream* function.

In the dynamic instances part of the PRManager in Fig. 3, the user can define any number of PRGroups. PRGroups are independent amongst each other and they follow the same interface specifications as the reconfigurable region. An example of three PRGroups is illustrated in Fig. 4. In the static Maxeler approach, the kernels and the connections of a manager are described in a Java constructor. Subsequently, a PRGroup in our approach can contain anything that an original manager Java constructor can contain. This way, we guarantee that the PRGroup can include different connections and one or more instantiated kernels.

A PRGroup must contain at least one kernel, as shown for PRGroup_N. This was the model when introducing the dynamic approach where a single kernel was assumed as a PRGroup to be reconfigurable. In addition, a PRGroup can contain a pipeline of kernels like PRGroup_0 in Fig. 4. In that example we can see that the kernels can have different interfaces between them, as shown in PRGroup_0 between

```
1  PRManager(PRManagerEngineParameters PRManagerEngineParameters)
2  {
3      super(PRManagerEngineParameters);
4      CustomHDLNode hdlNode    = new Icap_wrapper(this, "Icap_wrapper");
5      CustomHDLBlock icapModule = addCustomHDL(hdlNode);
6      icapModule.getInput("Icap") <== addStreamFromCPU("Icap");
7      KernelBlock k1 = addKernel(new PartialRegion (makeKernelParameters("PartialRegion")));
8      //More static kernels can be added here//
9
10     DFELink inStat_512 = addStreamFromCPU("inStat_512");
11     DFELink outStat_512 = addStreamToCPU("outStat_512");
12     k1.getInput("inStat_512") <== inStat_512;
13     outStat_512<== k1.getOutput("outStat_512");
14
15     void PRGroup("GroupA")
16     {
17         KernelBlock k1=addKernel(new SobelSolutionKernel(makeKernelParameters("
               SobelSolutionKernel")));
18         KernelBlock k2=addKernel(new BrightnessFilter(makeKernelParameters("
               BrightnessFilter")));
19
20         DFELink inRegion512 = addStreamFromCPU("Sob_in512");
21         k1.getInput("Sob_in512") <== Sob_in512;
22         k2.getInput("Bri_in512") <== k1.getOutput("Sob_out512");
23         DFELink outRegion512 = addStreamToCPU("Sob_out512");
24         outRegion512<== k2.getOutput("Bri_out512");
25     }
26 }
27 private static EngineInterface interfaceDefault()
28 {
29  EngineInterface engine_interface = new EngineInterface();
30  CPUTypes   type = CPUTypes.INT32;
31  int        size = type.sizeInBytes();
32  int     size512 = size*16;
33
34  InterfaceParam  N    = engine_interface.addParam("N", CPUTypes.INT);
35  InterfaceParam  k    = engine_interface.addParam("k", CPUTypes.INT);
36
37  engine_interface.setStream("inStat_512",    type,  N * size512);
38  engine_interface.setStream("outStat_512", type,    N * size512);
39  engine_interface.setStream("Icap",    type, k * size);
40
41  engine_interface.setDynamicStream("inRegion512",    type,  N * size512);
42  engine_interface.setDynamicStream("outRegion512", type,    N * size512);
43 }
```

Listing 1 Example Java code of the MaxJ language extension.

kernel_0 to kernel_1 and between kernel_1 and kernel_2. Moreover, a user can instantiate the same kernels in more than one PRGroup, as done for kernel_2 in PRGroup0 and PRGroup_1. A domain expert can create more complex PRGroups, like PRGroup_1 that is using kernels with different interfaces and streams in order to create a more complex PRGroup.

4.3 MaxJ Preprocessor

The here proposed language extension readapts the idea behind the Maxeler manager and introduces the PRManager that contains the extended functionality of a dynamic design. The reason for introducing the PRManager is that each project can only have one active manager file. Hence, to circumvent this issue, we allow multiple managers to generate different kernels and connections. To achieve that, we start from a main project and automatically split the main project into multiple smaller projects (one for static system and one for each PRGroup).

This is performed by a preprocessor that starts by detecting the PRGroups of the specified project, as shown in Fig. 5. For each one of the PRGroups, the tool creates a Maxeler project directory. Those Maxeler projects are only used by the tool to generate the RTL files of the corresponding kernels. The projects will use the connections defined in the manager interface with the extended function setDynamicStream (see Section 4.2), in order to describe the

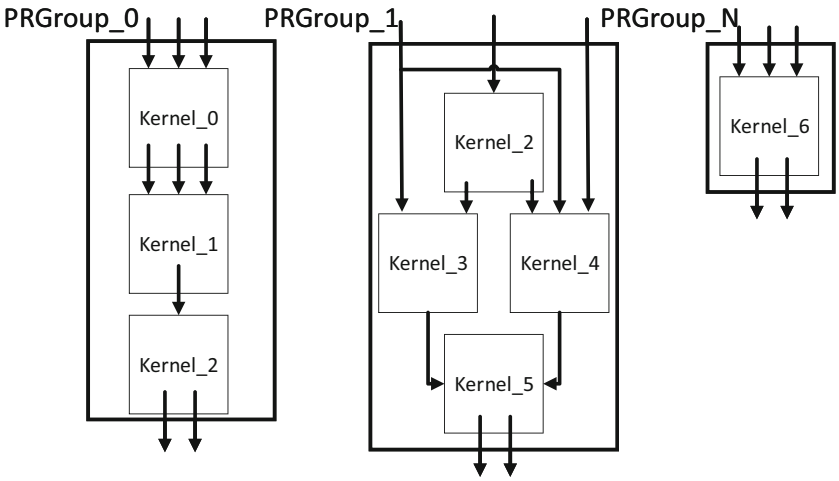

Figure 4 Example block diagrams of PRGroups. All depicted PRGroups have the same interface, (3 inputs and 2 outputs). A PRGroup can be a single pipeline like PRGroup0 or a more complex group like PRGroup1. PRGroup_N is a standalone kernel in this group.

interface of all the PRGroups. The setStream functions will set the connections with the static system.

In addition, the static part of the PRManager will be used to generate a static project, which will be the project which will later control the dynamic system. This ensures that the programmers code remains unchanged. The preprocessor concept allows for specifying the static system and all the kernels in one project that can be automatically compiled without touching the original Maxeler compiler or the FPGA vendor tools.

4.4 Front-end Overview

After the project generation, the preprocessor runs the MaxCompiler for all the projects until the end of the RTL generation step of the Maxeler toolflow, as shown in Fig. 5.

The main difference, compared to the original flow is that all the generated sub-projects will output possible error messages in one log file in the initial project directory in case of an error. At the same time, the static project will act as the root project as the user can manage the inputs

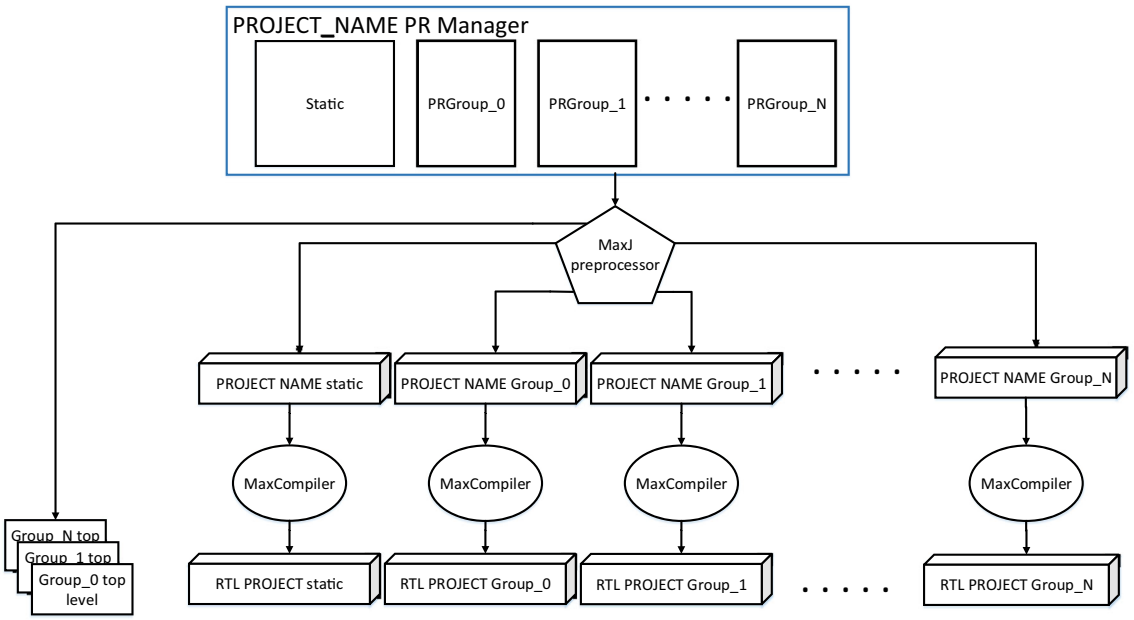

Figure 5 Front-end implementation flow. The input is the initial project for which the preprocessor generates the corresponding projects (project directories are depicted as 3d boxes). After the project generation, the MaxCompiler will process the MaxJ code and generate an RTL project version for each project generated by the MaxJ preprocessor. The preprocessor generates the final RTL top level entities of each existing group (left side of the Figure). The outputs of the front-end processing are the top level VHDL files for each group and the RTL projects generated by the MaxCompiler.

and outputs of the application from the static system and the loading of modules through ICAP.

The last step of the preprocessor is to generate the internal connections for the PRGroups in a VHDL top level file. This subprocess is illustrated in the left side of Fig. 5. The preprocessor will read the manager files in MaxJ. Then, it will register the connections of the kernels and it will create connections between the RTL generated kernels in VHDL. This will create the final PRGroup top level entity (in VHDL). All code transformations that are performed though our toolflow happen on the RTL code that is generated by the MaxCompiler and the entire process does not require any user interaction. A user of our flow will only interact with an HLS specification of the system (MaxJ).

4.5 Design Choices

Through our language extension, the user can generate not only partial modules, but groups of kernels that constitute a partial module. However, there may be kernels constituting PRGroups that can perform computations in any permutation order. Thus, if an application contains N kernels in different groups, generating every permutation will result in N! groups of kernels to be implemented. This solution is not scalable and, in some cases, it will be too time consuming during implementation phases for the FPGA.

Therefore, we foresee to generate individual chainable modules that require less implementations, if the configuration bitstreams of those kernels are relocatable. This section describes the implementation of individual kernels from PRGroups. The generation of each individual kernel is called *software design choices*, while the relocatability of the generated accelerator bitstreams will be described in Section 6.2.

We decided to provide users with more flexibility in terms of kernel combinations. An out-of-the-box approach is to allow the generation of all the kernels instantiated in PRGroups as individual partial modules for future usage (assuming relocatable bitstreams). The resulting partial modules can be chained directly by the run-time system. Those kernels will be called *unique kernels*. Our approach foresees to have only one physical implementation of a kernel (e.g. using multiple instances of the same kernel in the same or different PRGroups) and by using module relocation, a kernel can be executed at different locations on the FPGA.

The ability to reuse a physical implementation at different locations will not only improve CAD tool time but allow a system to compose stream processing pipelines from daisy-chained relocatable kernel bitstreams, provided that they are known at run-time. Though our flow, the generated partial modules, both kernel groups and unique kernels, will act as individual load-and-run modules. The above characteristic corresponds to the software design choices. However in hardware, we can allow additional design choices in terms of placement that will be managed transparently for the designer.

5 Architecture and RTL Implementation

When the Maxeler HLS compiler generates RTL code, it provides clear hierarchies with well-defined communication interfaces. We use this observation by our automated compilation flow to split the generated project into a static part and a set of accelerators. Additionally, the flow extracts the hierarchy that belongs to the static system (which will contain all the I/Os of the system) and the hierarchies that belong to the accelerators.

This section explains the static system design, the module implementation flow and the communication in more detail. For each distinct part of the design, we present the modifications that we applied to generalize the interface of our design.

5.1 Internal Communication

The static system and the accelerators generated by the MaxCompiler follow a predefined interface that is specified in MaxJ by a user. This interface is fixed to a Maxeler handshaking mechanism which is different for the input and the output, as depicted in Fig. 6. The generated kernels are using a chaining component which is generated by the Maxeler compiler and which handles the connections between kernels. This component contains a small FIFO and an I/O control block. In addition, we can observe that the Maxeler generated RTL kernels do not have a unified interface on the input and the output. However, in order to generate standalone partial modules, each accelerator should be completely independent of external instances (i.e. the chaining component) and each accelerator should have a common interface at the input and the output. The latter is important as we consider arbitrary chaining and placement of modules in the reconfigurable region. Thus, to

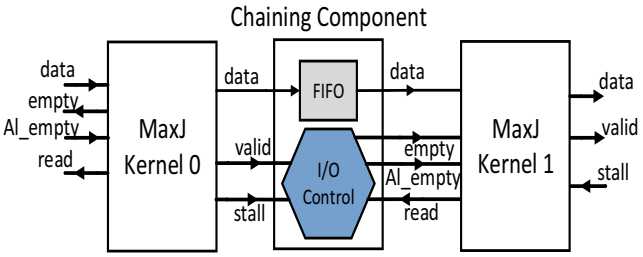

Figure 6 Original module chaining in Maxeler using a chaining component.

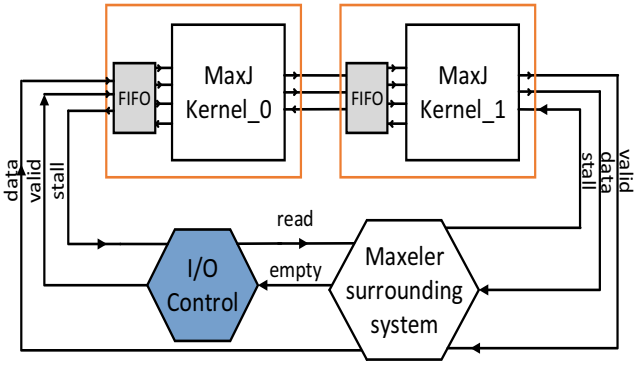

Figure 7 Modified kernels after the chaining module split. Kernels contain an integrated FIFO that existed in the initial chaining module, while its I/O control is placed in the static system.

introduce partial reconfiguration to the Maxeler platform, 1) a common I/O interface is required and 2) partial modules should be independent from external components.

In order to guarantee independency and relocatabillity amongst kernels, we created a common interface for reconfigurable modules. By separating the chaining component into two parts, we can integrate the FIFO in the kernel and keep the I/O control as part of the static system. This results in an architecture shown in Fig. 7. As we can see, the interface of the input and the output is exactly the same for each kernel wrapper. This is essential as it allows accelerators with the same interface to be arbitrarily daisy-chained by stitching relocatable bitstreams together. Those bitstreams can construct more complicated accelerator pipelines within the reconfigurable region.

Following the description on Section 4.5, the unique kernels and their hierarchies are saved in a specific directory for unique kernels. This directory contains the implementation-ready accelerators that will be available for daisy-chained placement by the run-time system. Additionally, the tool generates the PRGroup partial modules in their corresponding RTL version, by generating the top-level HDL code from the detected connections of the front-end processing (i.e. the MaxJ preprocessor). The mid-level process provides us with the number of resources needed for each PRGroup and each unique kernel. Those numbers are reported by the Xilinx tools and they are used to calculate the minimum resources needed for the region.

5.2 Static System

The initial static system provided by MaxCompiler contains an interface to the implemented accelerators. This interface is shown in Fig. 6. However, as mentioned in Section 5.1, the initial interface of the accelerators should be adapted to a more partial reconfiguration friendly interface in RTL. This has to be kept consistent with the I/O interface of the reconfigurable region.

After the MaxCompiler generates the full functionality of a Maxeler project (i.e. static system and accelerators), further modifications will be performed through the mid-level process. The static part will host the ICAP instance to allow loading the dynamic kernels on the FPGA device. However, we need to instantiate an RTL specific version of the reconfigurable region in the static system that cannot be instantiated in MaxJ. In addition, this region needs external input control components to handle the data streamed towards the reconfigurable region.

Figure 8 depicts a generated example system. Its architecture consists of the Maxeler surrounding system which contains I/O peripherals, the controls of the data I/O, and the main interface that transfers data from CPU to FPGA and vice versa though PCI-e or an existing memory interface. The system also instantiates a manager that contains a Custom HDL wrapper and one or more HLS generated kernels. Partial reconfiguration is performed through the Maxeler Interface, which is included in Maxeler's surrounding system which sends partial configuration bitstreams to the Internal Communication Access Port (ICAP) of the FPGA. The ICAP is instantiated inside a Custom HDL wrapper that forwards input data from the Maxeler interface to the ICAP (Fig. 9).

The reconfigurable region wrapper is a placeholder module having the same interface with the accelerators that will be instantiated by the run-time system. The MaxJ version of the reconfigurable region will be replaced during the RTL modification phase. The reason for this is that the region needs physical constraints that cannot be specified in MaxJ. Furthermore, the here presented flow aims at hiding

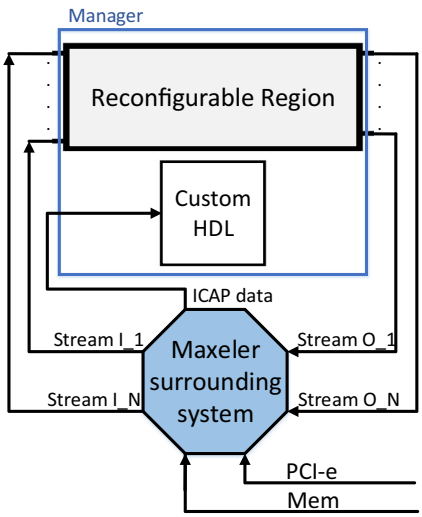

Figure 8 High level view on a Maxeler generated system, before modifications. Each Maxeler system contains a manager with all the instantiated kernels. In addition, the I/O drivers and controls are instantiated in the Maxeler surrounding system. This part remains intact by our toolflow.

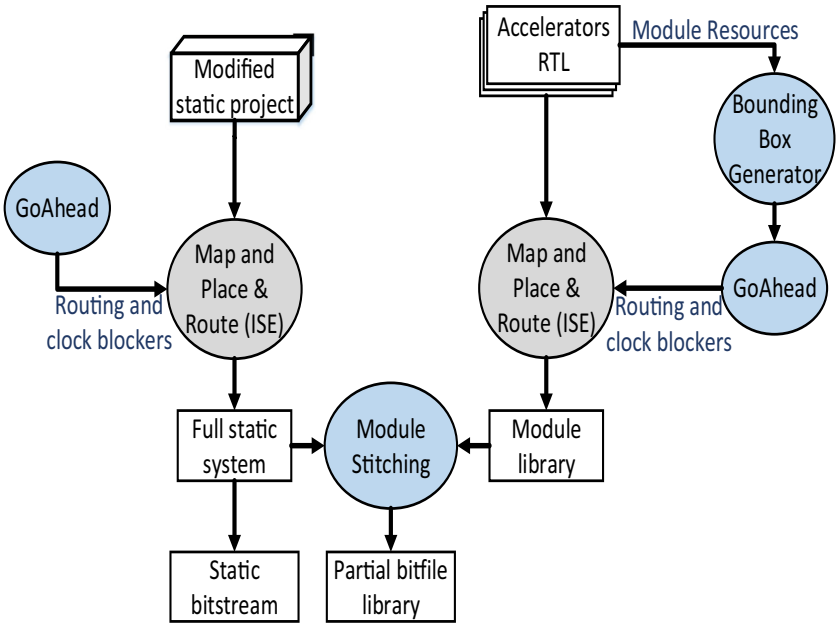

Figure 9 Toolflow diagram depicting the processing stages that are done after we modify the RTL code from the Maxeler Compiler. The left half of the figure illustrates the steps occurring while implementing the static part, while the right focuses on the partial module generation. External tools are marked in blue, while Xilinx tools are marked with grey.

all the FPGA low-level details. As a solution, we replace a MaxJ version of the reconfigurable region automatically with a project-specific reconfigurable region.

Our mid-level tool called *PR code generator* instantiates the region in the MaxJ generated RTL design. The instantiation occurs by copying the necessary files in the project directory and by setting location constraints for the region itself in the User Constraint File (ucf). Each stream towards the reconfigurable region contains an I/O control, as shown in the Fig. 7. The I/O control handles the I/O between two Maxeler instances. In our approach, this I/O control is shared among all the kernels that will be placed in the region for the specific stream (as in Fig. 7), instead of using one I/O control between each kernel (as in Fig. 6).

The reconfigurable region is implemented as a loopback device. By loopback device, we mean that the inputs of the specific module are routed across the fabric (i.e. the reconfigurable region) that it occupies, then performs a U-turn and follows a backward path towards the Maxeler surrounding system. The region interface is predefined at a 512-bit wide datapath, plus some extra handshaking signals, however the user can define the desirable width of the system, with a maximum of 512 bits of input data and 512-bits of output data. We selected a datapath of 512-bit as this allows for saturating a single DDR memory channel. In case that the user does not need all the bandwidth, our tool will automatically ground the remaining signals.

5.3 Mid-level processing

In our automatic toolflow, we run a Python script to generate the dynamic design. The script initially starts by preprocessing the MaxJ code and generating all the necessary projects. As depicted in Fig. 10 after the RTL generation by the MaxCompiler, the script automatically extracts the RTL code. This contains the IP cores given as netlists and constraint files, as indicated in Fig. 2. Then, the mid-level PR code generator processes separately the original Maxeler generated RTL code of the static part of the project and the partially reconfigurable kernels/accelerators in the form of PRGroups. In the case of the static system, we use an external tool (GoAhead) to generate the reconfigurable region and its location constrains. We will refer to this tool in more detail in Section 6.3.

For each detected PRGroup, the PR code generator creates an independent project directory, as done for each unique kernel. In those directories, the tool adds template files (such as FIFOs and controls) which include the necessary changes described previously in this section.

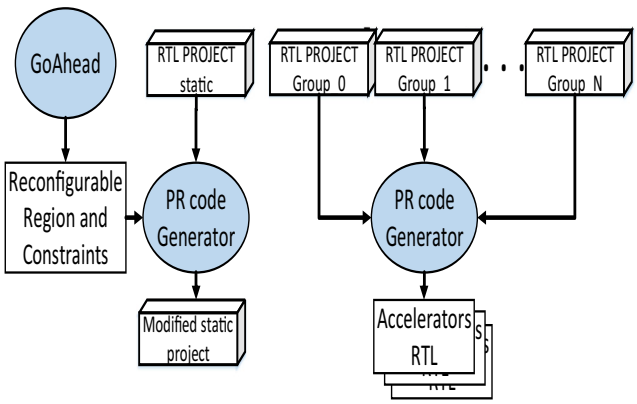

Figure 10 Mid-level processing. The RTL projects generated by the preprocessor and the Maxeler Compiler are modified to support partial reconfiguration.

Additionally, it modifies the generated files in order to include the changes in the RTL code. Those modifications are performed for both the static system and each kernel or PRGroup to guarantee independency.

The mid-level processing phase automatically performs changes for making all subparts of the implementation independent. This process outputs RTL projects, HDL instances and their resources that are ready for physical implementation. The physical implementation is done by the back-end process that handles all the low-level details of our flow.

6 Low-Level Specifications

The physical implementation of an FPGA-based system can take an extensive amount of time (in the range of hours) due to the long FPGA CAD tool time. This section describes how our toolflow allows building reconfigurable systems faster by design preservation and by enabling parallel compilation. Additionally, we provide a brief reference on the background tools that we use to create our dynamic system and explain the low level details of our approach.

The implementation process assigns an accelerator to a partial module and creates the static system. After the back-end processes, the static system will output a static bitstream, which will be the core execution binary of the dynamic project. On the other side, the modified accelerators will output a library of partial modules that correspond to the PRGroups and unique kernels. The back-end processes for the static part and the accelerators implementation will be analyzed in Sections 6.1 and 6.2, respectively.

6.1 Reconfigurable Region and Static System

The reconfigurable region is the key component that allows integrating run-time reconfigurable modules into the surrounding static system. Thus, low-level design details should be taken into account when physically implementing the static system. This in particular includes the data routing within the region as well as the clock routing. This subsection provides details on the physical implementation. Note that the interface of the reconfigurable region and the accelerators and the allocated resources of the region are covered in Section 5. The left half of Fig. 9 illustrates the flow for generating the static part of the dynamic project. After we generated the RTL version of the region, the placement constraints and blocker files, we run the ISE flow to generate a full bitfile of the static part of the dynamic system. The region is instantiated in the VHDL code during the modifications explained in Section 5.2. The blockers are files that constrain the routing of the clock signals and the data I/O and they are used only during the physical implementation phases. All the implementation files are generated by the GoAhead tool, which will be introduced later in this section.

A fully placed and routed region is depicted in Fig. 11 as a screenshot of Xilinx's FPGA Editor tool. The screenshot illustrates the physical implementation of the block diagram shown in Fig. 8. Figure 11 depicts a physically implemented partial region and the Maxeler surrounding system, which is entirely generated by the MaxCompiler. In addition, the physical implementation of the 512-bit loopback interface is placed on the left of the region and it is used to move data between the region and the Maxeler surrounding system.

For implementing the reconfigurable region, we manually floorplanned the system by taking into consideration where Maxeler maps specific components. This includes in particular the placement of the I/O cells for the PCIe and DDR3 memory connections. Following this, we observed that Maxeler does not use the corners of the device for implementing the FPGA's I/O infrastructure. Therefore, we place reconfigurable regions in the corners of the device.

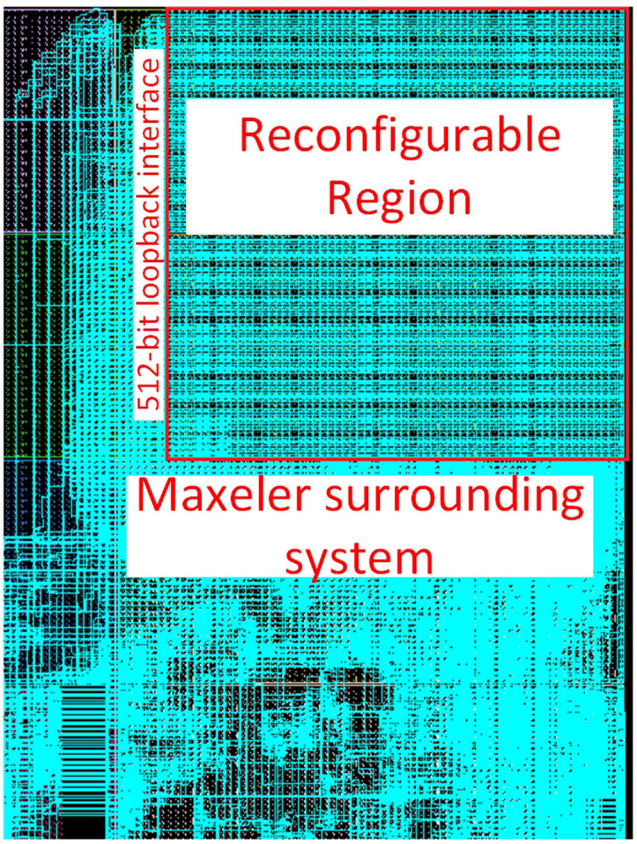

Figure 11 Empty reconfigurable region implementation (red boxed part). The interface contains 512-bit of input and 512-bit of output, plus some extra handshaking signals. The picture depicts the top-right part of the chip. We can also observe the predefined routing which is regularly structured within the region, while the rest of the system that surrounds it is freely routed by Xilinx vendor tools.

Note that this floorplanning can be used for future static systems implementations. Our tool comes with different reconfigurable region templates. Thus, a user can build a dynamic system without the need of manual floorplanning.

The reconfigurable region represents the physical implementation of the MaxJ kernel wrapper shown in Fig. 8. The reconfigurable region is empty except from some flip-flops, which are used to implement a pipelined loopback routing path. We also constraint the routing inside the reconfigurable region with blockers, both for internal signals and the clock. The blockers preoccupy FPGA routing resources and are used to guide the physical implementation of the Xilinx vendor tools (place and route). This is important to ensure that 1) the partially reconfigurable modules do not interfere or collide with any routing resources used by the static system, 2) all the I/O signals are routed through predefined wires (such that a wire will be the output of a specific signal in one kernel but the input of a consecutive kernel), and 3) the reconfigurable region provides a clock on exactly the same clock tree resources as used by the reconfigurable modules.

6.2 Module Generation

This section covers the physical implementation details to be considered when implementing reconfigurable modules. The implementation should ensure that an accelerator/PRGroup is reusable by other projects and that it can be loaded into a reconfigurable region. Thus, the same generalized routing for data and clock used in Section 6.1 must also be used for the physical implementation of the accelerators. However, partial modules require physical resources that have a corresponding resource footprint on the FPGA (see Section 6.3.2). Matching the resource footprint will guarantee relocation and replication of the desired partial module in different physical locations within a partial region. To achieve this, a major challenge will be to strictly restrict the full functionality of a partial module within a bounding box, based on the resources required by the accelerator.

The right half of Fig. 9 illustrates the automatic flow for the module generation. Starting from the modified generated RTL code of the kernel, we need to calculate exact module bounding boxes given the respective kernel's utilization (i.e. LUTs, DSPs, and BRAMs). After defining the kernel's bounding box, we need to generate placement constraints and blockers around the predefined position. Those blockers ensure that the module routing does not cross its bounding box borders. With the generated constraints, we can use the Xilinx toolchain to fully map and place and route the module.

In order to implement the interface routing of the module, we place flip-flops left and right of the module and leave "holes" in the blocker, as depicted in Fig. 12. In this

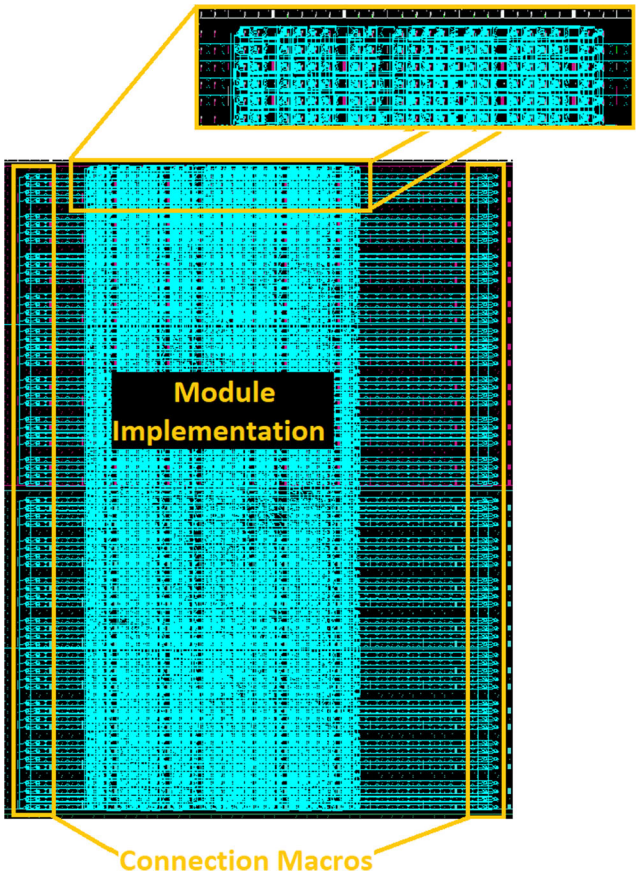

Figure 12 Fully placed and routed PRGroup depicting the connection macros and the interface connections. The connection macros substitute the connections and routing within the reconfigurable region. The zoomed figure shows that the routing is constrained not to cross the border (in this case the top border) as well as the interface wires on the left and the right side of the module implementation.

figure the module routing and interface is being depicted in the zoomed subfigure. The RTL description of the unique kernels and PRGroups is instantiated between the interface flip-flops. The interface flip-flops are called *Connection macros* and they are used as anchor points during the physical implementation of the common interface of the modules and the region to guarantee relocatability of the generated partial modules. This process, can be run in parallel, in order to generate the corresponding physically implemented instances. Because there is no dependency between the PRGroups and the unique kernels, it is possible to run all PRGroup implementations and the static system implementation in parallel.

The aforementioned process results in several implementations of fully implemented blocks of unique kernels and PRGroups. Those blocks can be placed at different positions that are pre-calculated initially within the reconfigurable region. The result of the process is a final static configuration bitstream and we can generate partial bitstreams using either bitstream manipulation with the BitMan tool (see

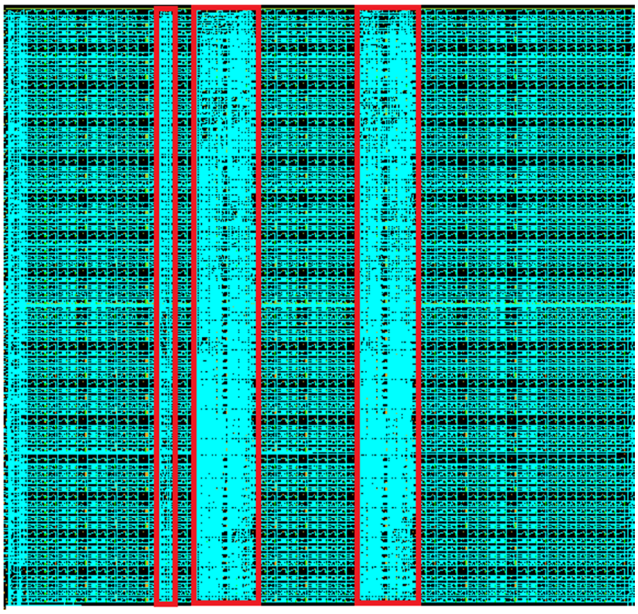

Figure 13 Stitching of reconfigurable modules in a reconfigurable region in a pipeline fashion, following our dynamic approach.

Section 6.3) or by using the differential bitstream methodology available by Xilinx. In the latter case, we need to place the netlist of a reconfigurable module (including place and route information) at the target position of our static system (with the help of the GoAhead [18] tool). Both techniques allow module relocation and replication. Figure 13 depicts 3 reconfigurable modules, which are placed in the reconfigurable region. Bitstream manipulation (BitMan [19]) allows for rapid stitching of modules at run-time without running any of the tools of the vendor Xilinx and relocation at netlist level (using the GoAhead tool) allows static timing analysis using Xilinx vendor tools. With this feature, we can guarantee that any system created through bitstream manipulation at run-time will meet timing (including both setup and hold time).

The partial bitstreams can be saved in a library of stitchable kernels (given as partial bitstreams). Our flow can automatically generate multiple physical implementations of the same module to incorporate the heterogeneous resource layout of logic, memory and arithmetic block columns (CLBs, BRAMs and DSPs). This allows for a tighter placing of modules. The combination of implementation alternatives and placement position alternatives constitute the *hardware design choices*, that are adding to the software design choices, described in Section 4.5. Each PRGroup will also generate a partial bitstream. In case a PRGroup contains multiple kernels, its kernels will be outputted as partial bitstreams as well and they can be used for building combinations of accelerators at run-time, should it be needed by the user. Of course, a user can always define a new PRGroup through the MaxJ code and follow the whole toolflow to physically generate a new PRGroup; however, this will be time consuming as the tool needs to generate a new physical implementation. By generating the accelerators individually, we offer a solution that can be generated in milliseconds rather than hours.

When all bitfiles and placement positions are generated, we load the kernels in user-definable order into the reconfigurable region. A user is also able to replicate or relocate existing modules. For that purpose, we save the initially calculated placement positions in the design phase in order to incorporate the placement of implementation alternatives. This is automated by BitMan that adds corresponding metadata to the original Xilinx configuration bitstream.

In summary, one of the main contributions of this work compared to full static implementations, is that the flow can be parallelized. This is important as the mapping and the routing tools need significantly more time with rising complexity. Thus, splitting the design into distinct parts (partial modules and static part) can save CAD tool time when executed in parallel. Moreover with our flow, every modification of the static system or an addition of a kernel or a PRGroup will be done independently, without the need of recompiling and re-implementing (including place and route) the whole architecture. This is because changes in the static system or a module will not interfere with other parts of the system, hence allowing for recompiling only the parts of the system that are changed.

Having introduced the implementation flow, we will now take a closer look into the tools that we use in the background to hide the low-level FPGA specific details.

6.3 Customized Tools

The FPGA vendor Xilinx provides tools that allow for implementing run-time reconfigurable systems. However, this flow implements partially reconfigurable modules as an increment to a static system. This means that a module can only work in its particular static system and that any change to the static system requires a re-implementation of the corresponding modules. In order to provide systems with more flexibility, where modules can be implemented independently to the static system and where modules can be relocated across multiple different static systems, we are using a chain of existing academic tools.

6.3.1 GoAhead

GoAhead is a tool that is used to create all the components for the system's reconfigurable design. GoAhead provides floorplanning capabilities, communication infrastructure, and constraints generation that are required for the physical implementation. GoAhead can be controlled by either a

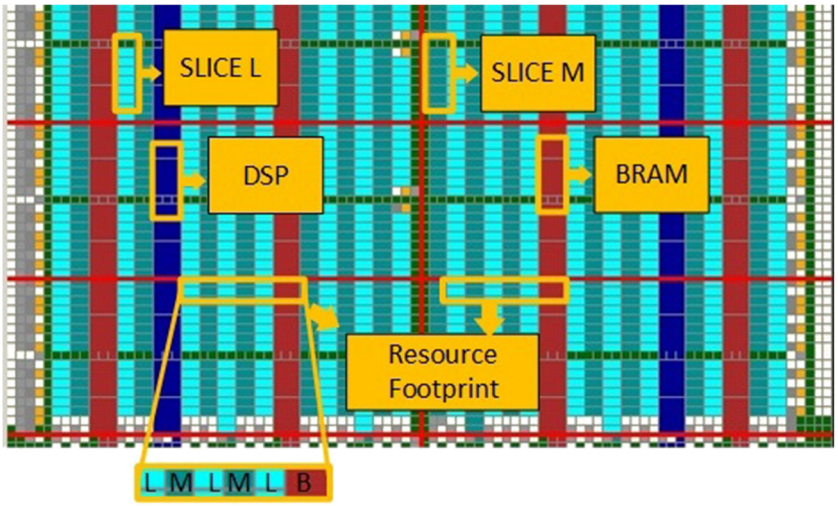

Figure 14 GoAhead GUI displaying device primitives and an example for the definition of a resource footprint.

GUI or through scripting. We use the scripting interface in order to generate module bounding boxes or reconfigurable regions, including all VHDL code templates and physical constraints (for controlling the place and route step to comply with the partial reconfiguration rules). A GUI (Fig. 14) can be used for debugging, floorplanning purposes, and module placement (at netlist level).

6.3.2 Bounding Box Generator

In order to search automatically for placement positions for partial modules, we used the tool *Bounding Box Generator (BBG)* [20], which checks the reconfigurable regions based on different resource footprints. The resource footprint is the relative layout of the logic primitive columns that a module occupies, modeled as a string. The symbols of the string represent the exact sequential order of primitive columns (within a partial region or a module). Figure 14 provides an example of a resource footprint. Each resource is shown with a different color in GoAhead such as red for BRAMs *(B)*, dark blue for DSPs *(D)*, light blue for SLICELs *(L)* and cyan for SLICEM *(M)*. The tool checks for possible placement positions and as the figure depicts, in this specific case, the example module has 2 columns of possible placement positions horizontally (indicated by the resource footprint, shown in the Fig. 14).

6.3.3 BitMan

We use the tool *BitMan* to modify our design at the bitstream level. BitMan supports functions that include module placement, replication, and relocation of FPGA bitstreams. BitMan can work as a standalone tool, but also offers an API to place, relocate, and replicate modules directly when running applications on the Max3 system. BitMan supports Virtex-6, all 7-Series, and UltraScale FPGAs from the vendor Xilinx.

7 Results

This section presents our results in terms of additional resource requirements and timing overhead compared to the initial non-reconfigurable Maxeler generated architecture. In addition, we present design time metrics, as well as the time that the tool needs to reconfigure the device.

7.1 Experimental setup

For our experiments, we used a Max3 Workstation which provides a large Xilinx Virtex-6 XC6VSX475T FPGA which is connected to the host computer via PCIe. The FPGA is surrounded by 24GB (upgradable to 48GB) of DDR-3 memory and the host CPU is an Intel(R) Core(TM) i7-2600S clocked at 2.80GHz.

To demonstrate the benefits of our flow, a video/image stream processing application was implemented where various modules can be arbitrarily chained to form more complex acceleration pipelines. A video of the flow and the system in action is available at [21].

Our module library consists of 8 image processing functions. Those are Brightness correction, Sobel edge detection, RGB-to-Greyscale, Skin Color Detection, Gaussian blur, a Mean, a Minimum, and a Maximum value filter. In addition we have implemented 3 PRGroups, containing a Mean with a RGB filter, Brightness filter with a Skin detection filter, and a Min and Max filter. All of those functions are generated entirely by the Maxeler compiler from MaxJ code. Figures 15 and 16 depicts an FPGA-editor screenshot showing placed and routed kernels and routed PRGroups.

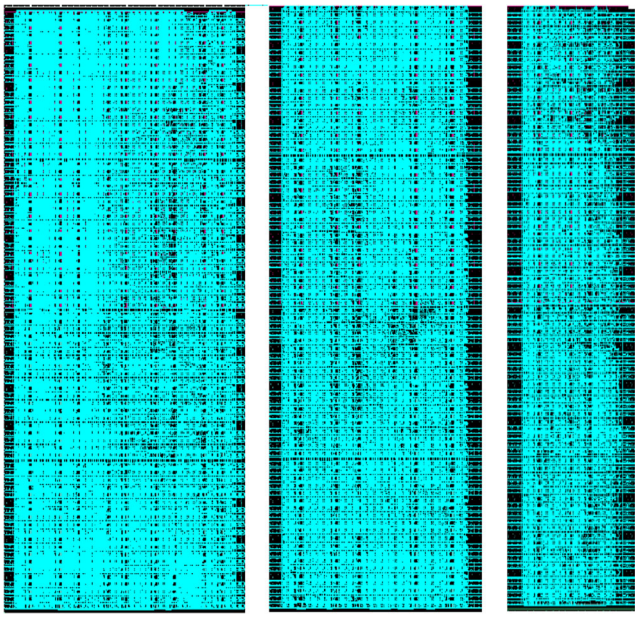

Figure 15 Fully implemented PRGroups, as shown in Xilinx's FPGA editor tool. The illustrated kernels are (l.t.r.) Mean with a RGB filter, Brightness filter with a Skin detection filter, and a Min and Max filter.

It can be seen in Fig. 12, that the interfaces on both sides of each module use the same equivalent wires, as previously depicted in Fig. 12, which is the key property for module chaining.

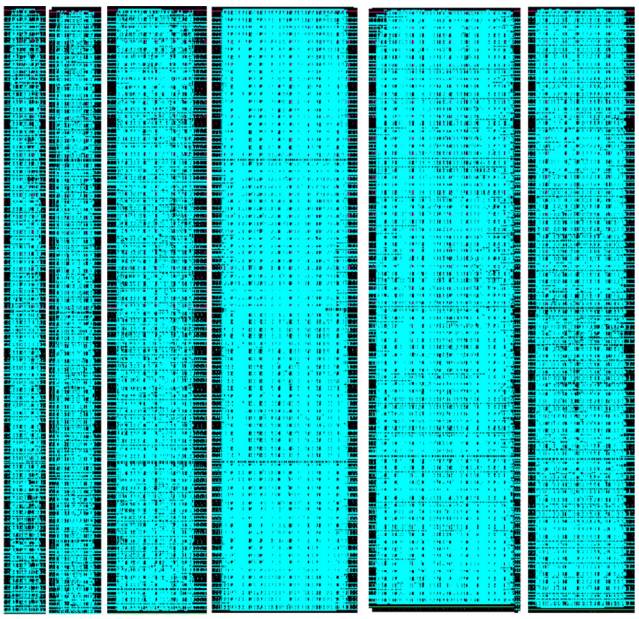

Figure 16 Fully implemented kernels, as shown in Xilinx's FPGA editor tool. The illustrated kernels are (l.t.r.) Brightness correction, RGB to GreyScale, Skin Color Detection, Gaussian Blur, Sobel ED filter, and Mean Filter.

7.2 Resource Utilization

The utilization of the generated kernels is shown in Table 1. The overhead that occurred due to the modifications, which are described in Section 5 is included in those numbers. The additional implementation cost of each kernel is about 441 LUTs (mainly used for 512-bit wide and 32 positions deep FIFOs which are implemented by distributed memory). Note that in Table 1, the modules do not contain DSP primitives, however, we fully support relocation of DSPs.

The static system consists of 9375 LUTs, 22 BRAMs and has one input and one output of 512 bits for the image operator's I/O and one input for the ICAP Custom HDL module. In the implemented example, the reconfigurable region offers 28800 CLBs, 128 BRAMs, and 112 DSPs and it is placed in the top right corner of the chip. This placement was chosen because the static part and the connections are placed in the center of the device next to the PCIe interface.

As a reference, we implemented a non-reconfigurable full static (FS) design that provides all the aforementioned kernels in parallel. This would correspond to a system that runs all the aforementioned kernels in parallel. In this system, the input can be streamed through all possible kernels and back to the host machine, if needed. As an alternative approach, a user can implement multiple different projects and reprogram the device with the project that contains only the filters needed, which, firstly, takes time and secondly, in case of switching accelerators in most real-world streaming applications (e.g., a video processing platform), the operation of the system would be interrupted for a significant amount of time. Additionally, there are cases where a user may need to run a combination of kernels in a pipelined fashion. However as the number of possible kernel choices grows, it is not feasible to create a design for every combination of kernels that can be combined. Moreover, some filters could be mutual exclusive to each other (e.g., Mean filter versus Gaussian filter), which implies that hosting them both in a static solution would result in an underutilized FPGA implementation.

Additionally, our framework offers the freedom to load any kernel or any number of the available kernels, as long as they fit into the reconfigurable region, and allow loading and resetting in less than ten milliseconds. This is possible while keeping the FPGA RAM active during reconfiguration and idling only the reconfigurable region during the process. The reconfigurable region used in our case study provides only 10% of the FPGA resources and much more resources can be allocated for the reconfigurable region, if needed. Please note that a user can decide to only implement parts of the system to be run-time reconfigurable, while leaving other accelerator kernels static (e.g., if those kernels are being used statically).

Table 1 Resource utilization of generated PRGroups and Unique kernels.

Kernel	LUTs	BRAMs	DSPs
Brightness correction	4444	1	0
RGB to Grayscale	6814	1	0
Sobel ED filter	26482	17	0
Gaussian blur filter	12659	17	0
Mean filter	12678	16	0
Skin Color Detection	9821	14	0
Minimum	4582	1	0
Maximum	4556	1	0
Mean + RGB	19292	17	0
Brightness + Skin	14358	15	0
Min + Max	9138	2	0
Example full static (FS)	92405	112	0

7.3 Compilation and Configuration Time

The Maxeler Compiler that translates MaxJ to VHDL works relatively fast. For example, the tool provides the RTL code description for all the example cases (including the FS mentioned in the previous subsection) in between 10 to 20 minutes. After that step, the FPGA vendor tools carry out the entire physical implementation until the final configuration bitstream. The results for the compilation time of our approach are shown in Table 2. The time metrics include the time needed for RTL generation from Maxeler, BBG, GoAhead and the whole toolflow from RTL synthesis to bitfile generation. Thus, the tools have to implement 11 individual (unique kernels plus PRGroups) and significantly smaller designs and another one for the static part. All 6 distinct parts (i.e. static system and 5 PRGroups of which 3 PRGroups contain 2 kernels each and 2 PRGroups contain 1 kernel each) together contain about the same logic, as compared with the FS design, if they are combined. On the contrary, the FS design needs 74 minutes from the RTL generation to the final MaxFile, or more than 25% additional time. Please note that the compilation times for the partial modules and the static system include a maximum cost of 10 seconds which is needed for the VHDL code processing and project generation.

The reconfiguration time of all our modules is listed in Table 2. As listed, each kernel takes from 3 to 6 ms for configuration by the ICAP instance on the device. After the configuration is done, we can execute the function loaded by sending the image data from the DDR memory or PCIe. This execution step remains the same as in the static Maxeler approach. Each reconfigurable region can

Table 2 Time needed for reconfigurable module generation, the static part of our reconfigurable module and the full static design.

Kernel	Compilation time (min)	Reconfig. time (msec)
Brightness correction	25	3
RGB to Greyscale	27	5
Mean filter	30	6
Gausian blur filter	26	5
Sobel ED filter	43	5
Skin Color Detection	32	5
Minimum	27	3
Maximum	26	3
Mean + RGB	39	4
Brightness + Skin	38	4
Min + Max	31	4
Static Part	56	–
Resetting bitstream	1	10
Example full static (FS)	74	–

Compilation time does not include the RTL generation time. The FS design includes only the Maxeler compilation time, as it is full static

be resetted individually by generating a resetting bitstream for each region and loading this file through the interface C-code of the Maxeler project.

Finally, module relocation at run-time is done by the BitMan tool.

For the stitching of the aforementioned kernels, BitMan needs 10 to 20 seconds to generate the final relocated bitstream for each possible kernel. In many practical systems, this process could be carried out once and stored for future configuration processes.

The current system is clocked at 100 MHz, which is the default clock frequency set by Maxeler, hence providing a peak performance of 6.4 GB/s on our 512-bit wide datapath. This performance can be achieved by all pixel operators implemented for our example case. The system does include a small penalty in latency for the single kernel designs, as the reconfigurable region itself includes a pipeline latency of a maximum of 5 clock cycles. For even higher throughput, multiple reconfigurable regions may be used in parallel (if permitted by the application). Considering memory-to-memory streaming, it would require three reconfigurable regions working on parallel to fully utilize the aggregated memory bandwidth available through three DDR-3 memory channels available on the used MAX3 system.

7.4 Existing Work

Compared to the existing work on applying partial reconfiguration techniques on the Maxeler platform, as shown in the results of [12], the configuration time is significantly slower than ours, as our bitstream with the highest utilization (resetting bitstream with an area of 28800 LUTs) needs only 10ms to configure the reconfigurable region, while for an area of 11764 LUTs the authors in [12] need 400ms. The main reason for our faster reconfiguration time is the fact that we use a faster configuration interface (ICAP) that runs consistently at its 400MB/s maximum configuration throughput, while in [12], the authors use a different interface that is operated slower on Maxeler systems.

However, the most important difference is that the authors in [12] have not considered placing multiple modules in a single region. This work suggests chaining modules at compile time, but that would be prohibitively slow for run-time adaptive systems such as database acceleration where the exact chain of the modules is only known at run-time. In addition, [12] does not consider automatic placement, which is essential when introducing a low-level implementation for non-FPGA experts. Lastly, the authors mention that they use the Xilinx PR flow for their proposed implementation. Thus, each kernel is therefore hardwired in a region and cannot be replicated or relocated. This makes sharing for more throughput much more difficult.

8 Conclusion

This paper presented a complete framework around the Maxeler platform that extends the current full static to a dynamic dataflow approach. The paper presents a top-down flow of our approach, starting from a language extension that models the dynamic aspect of a system. The presented work processes the Maxeler's generated code and the changes needed to implement a reconfigurable system directly from the output of the MaxCompiler. Additionally, clock speed was not found to be impacted by the presented work, while latency is only impacted by 5 clock cycles. Moreover, the reconfiguration time is significantly lower than the time to reprogram the device with a different full bitstream. Moreover, the whole PR design complexity is entirely hidden from the user, as in the dynamic flow, users will only have to write MaxJ and C code, exactly as it was performed in the original Maxeler case for the full static implementation. Last but not least, compared to the Xilinx vendor PR tools, the proposed flow allows for multiple kernels/functions to be placed in the same reconfigurable region and the proposed flow guarantees independency not only within the same project, but also across multiple projects, given that we enable reusability of the generated modules through a common physical interface.

The here presented flow can be used with other applications that can benefit from our dynamic HLS approach. Every application that contains run-time mutually exclusive applications can potentially be implemented and optimized using our approach. In addition, applications that can not fit in one device can also benefit from partial reconfiguration. For example, instead of cascading an application across different FPGAs using Maxeler's MaxRing protocol, our approach enables a time-multiplexing of a single FPGA. In this scenario, an application needs to be split into different FPGA partitions and could be executed on one physical FPGA using our dynamic approach for time multiplexing (e.g. by using external RAM to buffer data between partitions). The above statement is more critical if we consider the comparison between the PCIe and the on board memory bandwidth, which are, theoretically, 4GB/s (8 data transfer lanes) and 36 GB/s (3 DDR3 memory channels of 12 GB/s each) respectively. Thus, such an application can be split in kernel A and kernel B, which in turn will be implemented as 2 partial bitstreams. With this, we can load partial bitstream A, save the results in the on-board memory and replace module A with module B.

Our work demonstrated that building a partially reconfigurable system could be performed by domain experts using

only MaxJ and C code. We implemented our flow on a Xilinx Virtex 6 FPGA (i.e. the device used in Max3 systems). The most recent release from Maxeler is Max5 based on Xilinx UltraScale+ devices. As the Maxeler's toolflow has not changed significantly for the newest Max5 platform and because all the external tools that we used have been tested to work with UltraScale+ devices, porting the here presented approach should be straight forward. This is planned as future work, with the availability of Max5 hardware.

Acknowledgments We thank the university program of Maxeler Technologies for supporting our research.

References

1. Sunway taihulight supercomputer (2017) https://www.top500.org/system/178764.
2. DeHon, A., et al. (1999). Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *ACM/SIGDA*.
3. Baidu (2015) https://www.bdti.com/InsideDSP/2015/01/22/Xilinx.
4. Amazon f1 instances, https://aws.amazon.com/ec2/instance-types/f1/.
5. Alibaba cloud services, https://www.alibabacloud.com/.
6. Microsoft azure, https://azure.microsoft.com/.
7. Wirbel, L. (2014). Xilinx SDAccel.
8. Stone, J.E., et al. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*.
9. Sohanghpurwala, A.A., et al. (2011). OpenPR: An open-source partial-reconfiguration toolkit for Xilinx FPGAs. In *(IPDPSW)*.
10. Beckhoff Christian et al. (2012). GoAhead: A partial reconfiguration framework. In *(FCCM)*.
11. Jensen, J.J. (2012). *Reconfigurable FPGA accelerator for databases*. Master's thesis University of Oslo.
12. Cattaneo, R., et al. (2013). Runtime adaptation on dataflow HPC platforms. In *2013 NASA/ESA AHS 2013*. Torino.
13. Lindtjorn, O., et al. (2011). Beyond traditional microprocessors for geoscience high-performance computing applications.
14. Smaragdos Georgios et al. (2014). FPGA-based biophysically-meaningful modeling of olivocerebellar neurons. In *FPGA*.
15. Wenlai Zhao et al. (2016). F-CNN: An FPGA-based framework for training convolutional neural networks. In *ASAP*.
16. Maxeler app gallery, http://appgallery.maxeler.com/.
17. Technologies, M. (2014). Multiscale dataflow programing.
18. Beckhoff, C., et al. (2013). Automatic floorplanning and interface synthesis of island style reconfigurable systems with GoAhead. In *ARCS* (pp. 303–316): Springer.
19. Pham, K.D., et al. (2017). BITMAN: A tool and API for FPGA bitstream manipulations. In *DATE* (pp. 894–897).
20. Grigore, N.B., et al. (2018). HLS enabled partially reconfigurable module implementation In ARCS (pp. 269–282).
21. Example video, https://youtu.be/vgVNCVeqs8M.