Runtime Adaptation on Dataflow HPC Platforms

R. Cattaneo[1], C. Pilato[1], M. Mastinu[1], O. Kadlcek[2], O. Pell[2], M. D. Santambrogio[1]

[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy,
{*rcattaneo,pilato,santambr*}*@elet.polimi.it, matteo.mastinu@mail.polimi.it*

[2]Maxeler Technologies Ltd., London, UK,
{*okadlcek,oliver*}*@maxeler.com*

***Abstract*—We are facing an ever growing quest for performance in High Performance Computing (HPC) systems. The growing concerns for the power budgets and overall deployment costs required to run these systems are opening new ways to novel high performance computing platforms. New paradigms and architectures are being developed to tackle these challenges.**

In this context, FPGA-based HPC platforms employed to accelerate algorithms expressed as data flow programs are a promising paradigm. One traditional limiting factor of FPGA technology is that the ever increasing complexity of the applications might require the designer to switch to a bigger device or, conversely, the same device might be underutilized due to difficulties at sharing the available logic. Partial Reconfiguration is the standard technique to overcome such limitations.

This paper presents the research work done during the technology transfer to extend the Maxeler design flow to efficiently support Partial Reconfiguration (PR). In this work we focus on the design and development of a methodology to support the PR feature in the Maxeler design flow, a commercially successful FPGA-based HPC platform, showing the advantages of such an approach on the resulting platform.

I. Introduction

Nowadays computing systems are becoming more and more complex, with heterogeneous solutions that integrate CPU and Field Programmable Gate Arrays (FPGAs) in the same device. In particular, FPGAs [1] [2] allow hardware designers to change the implemented functionality after being manufactured. This fundamental difference with Application Specific Integrated Circuits (ASICs) makes FPGAs perfect platforms to emulate real hardware circuits, accelerate applications' specific portions of code, prototype complex ASICs and improve reliability of the overall design. In a traditional setting, the logic realized on FPGA-based systems is implemented as a single configuration, meaning that the application cannot change the circuital behavior while it is running. In this kind of applications the entire device's configuration is determined at compile-time and does not change throughout system operation. Due to the continuous improvement of the technology in terms of programmable components' density and capacity they are becoming more and more employed in different types of applications, spacing from building entire SoCs to employing them as co-processors in heterogeneous systems. For instance, Cray [3] proposed the XT5h super computer, based on a combination of AMD Opterons CPU with an FPGA-based Reconfigurable Processor Units (RPUs), while Intel in 2010 proposed a system where a CPU and an FPGA are placed in the same die [4].

In this context, heterogeneous (imperative and dataflow) hardware platforms are in great need. One example of such platform is provided by Maxeler Technologies Ltd. [5], a company that produces High Performance Computing (HPC) systems that use the dataflow paradigm in order to speed up the computation.

The main contribution of this work is the design and the implementation of a complete solution to support Partial Reconfiguration in the Maxeler design flow, along with the required modifications to the original toolchain. Adding the *PR* to MaxCompiler can potentially bring to very exciting implications, both in the academic and in the commercial world, e.g., allowing the application designers to exploit PR in an easier way, making use of the well tested and reliable MaxCompiler. To validate the work, a video filtering analysis application will be proposed, consisting of an edge-detector application, exploiting the newly supported Partial Reconfiguration.

The rest of this paper is structured as follows. Section II introduces the Maxeler architecture and the results that have been achieved so far with its adoption in hardware acceleration. Section III describes the implementation of the proposed solution that is then detailed in Section IV and evaluated with a test case in Section V. Section VI goes through the related works and, finally, Section VII explains the limitations of the proposed approach and outlines the future directions of work.

II. Maxeler Architecture Background

This section provides an introduction to the Maxeler Architecture. It describes at first the general structure of a Maxeler computing platform and how it has been *extend* to support the partial reconfiguration. The chosen architecture is basically composed,by two main components: the *CPU*, namely *the host*, (for running the software part of the application) and one or more *DataFlow Engines* (*DFEs*) (for running the hardware part of the application), connected each other through a PCI Express link ($2GB/s$ in each direction), as shown in Figure 1. *DFEs* are usually realized using an FPGA located on custom Printed Circuit Board (PCB) manufactured by Maxeler. A *DFE-board* is made of one (or more) Xilinx *FPGA*s. The

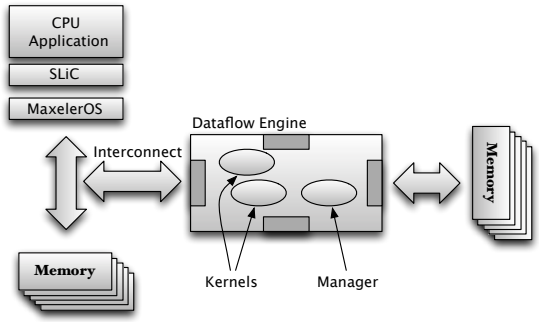

Fig. 1. Maxeler architecture diagram, with most relevant hardware/software components and buses.

FPGA are programmed using the dataflow computing model, where computations are implemented as a directed graph of operations, granting huge speed-ups compared to a generic software implementation. There can be more than one logical DFEs, connected together via high-bandwidth interconnections (*MaxRing*), which allows applications to scale linearly with multiple DFEs in the system, supporting at the same time full overlap of communication and computation. The presented work has been developed on a *MaxWorkstation* [6] but this is not a limitation and it is easily deployed also on the other available platforms. MaxWorkstation is a computing system running the *Centos 5* Linux distribution with an Intel i7 processor and a MAX3 board. As shown in Fig 2, the Maxeler MAX3 board is composed of two Xilinx Virtex-6 FPGAs (an *IFPGA* and a *CFPGA*), 24 GB SDRAM DDR3, an high performance memory (64 bit) controller that gives the theoretical maximum read/write bandwidth of 38.4GB/s. The two FPGAs are connected by means of an Inter-FPGA

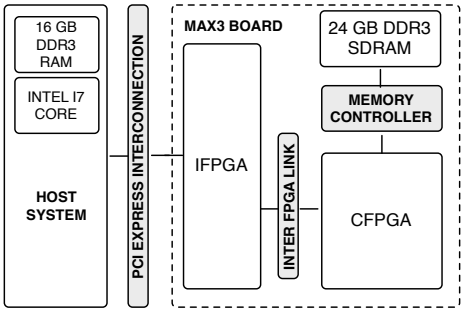

Fig. 2. A complete overview of the MaxWorkstation architecture

link. The two FPGAs have a different roles: the Interface-FPGA (IFPGA) is responsible for the PCI Express link management and other internal management tasks, while the Compute-FPGA (CFPGA) performs the user-programmable computations. When the MaxWorkstation is switched on, the IFPGA is configured with the bitstream contained in a flash memory inside the MAX3 board, while the CFPGA is initially empty. The bitstream used for configuring the CFPGA is stored externally to the MAX3 board on the host hard disk. The bitstream for configuring the CFPGA is stored the so called *MaxFile*, which is generated when a DFE application is compiled. The configuration of the CFPGA is initiated by the host CPU using the so called *SLiC* runtime interface. SLiC in turn calls low-level routines provided *MaxelerOS* hardware abstraction layer to transfer the bitstream to the IFPGA over *PCI* Express link using direct memory access. The IFPGA is managed by a finite state machine, which controls the configuration process. Upon receiving the command to configure the CFPGA, the state machine transfers the bitstream from the incoming PCI express port to the SelectMAP on the CFPGA, initiating the actual configuration of the CFPGA. The PCI Express link connecting the host CPU and the IFPGA is a *control stream* internal to the Maxeler infrastructure and not accessible to users. Within such a scenario, also the partial bitstreams used for configuring the CFPGA have been made part of the*MaxFile*. When the application is compiled, the resultant binary contains a buffer (similar to a C array) which stores the necessary data. The configuration of the CFPGA is managed, as said, by the host CPU using the *SLiC interface*: by exploiting MaxelerOS routines, the daemon which runs on the MaxWorkstation takes sequentially with direct memory access 64 bits chunks of the bitstream, and they arrive trough the *PCI* Express link to the IFPGA, as shown in Figure 3. The IFPGA is managed by a finite state machine, which

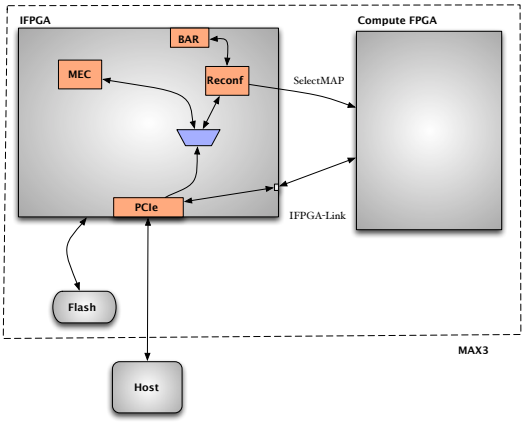

Fig. 3. The *Reconf* area represents the state machine responsible for configuring the CFPGA. The SelectMAP is used to access the configuration memory to updated it with both complete and partial bitstreams.

manages the configuration process. Thus, the role of the state machine is of managing the configuration, allowing the passage of the bitstream and signaling possible errors in the process. When the configuration has finished (the `DONE` signal in the CFPGA is set to high) the state machine asserts the CFPGA reset for a few cycles and then signals to the daemon that the configurations has successfully ended. At that point, *max_open_device()* returns and the application continues with the rest of the code. Within this context, working with a complete bitstream or partial ones is not going to vary the way in which the configuration is going to happen. The partial reconfiguration will be *physically* implemented via the download of the proper partial bitstream while the correctness of the process will be guarantee by the host code executed on the host CPU via an extended version of the *SLiC* interface in order to issue the partial reconfiguration of the device, as described in Section III. Programming an application for a Maxeler platform requires to write two different types of code:

- The *host code*, which runs on the *CPU*;
- The *dataflow code*, which runs on the *DFE*.

The host code can be written in either C/C++ or FORTRAN, while the dataflow code must be written in Java. A logical DFE is composed of two main elements:

- *Kernels* are hardware data-paths implementing the arithmetic and logic computation needed within the algorithm. They perform all the actual computation inside the DFE;
- *Manager* comprises all the logic that manages the data flow between Kernels and off chip I/O in the form of streams. Its goal is to manage the connections among the Kernels, and among the Kernels and the host application.

One of the peculiarities of the Maxeler platform is that the high-level language *MaxJ*, an extended version Java, is used to describe the hardware part of the application. This aspect outlines one of the greatest advantages of this platforms: one of the reasons why hardware acceleration on *FPGA* is not as popular as hardware acceleration based on *GPGPU*, is because programming an *FPGA* is usually a complex task, due to the difficulty intrinsic to writing code in some Hardware Description Languages (HDL). The usage of a well known high level programming language, such as Java, makes the task a lot easier for many programmers. A tool developed by Maxeler, *MaxCompiler*, includes tools to support all three steps: the Kernel Compiler, the Manager Compiler and a software library (accessible from C or Fortran) for bridging between hardware and software. The MaxCompiler transforms user kernels into low-level hardware and generates a hardware dataflow implementation (the .max file) which the developer can link into their CPU application using the standard GNU development tool chain.

III. Proposed Implementation

The proposed implementation, along with the design choices at the basis of our work is given in subsection III-A. We also describe the modifications that have been applied to the MaxCompiler to support the new design flow in subsection III-B.

A. Design Choices

Since all the dataflow computation is done by the kernels, it has been decided that the most suitable DFE parts where to apply the PR were the kernels. On the other side, it is not profitable to apply the reconfiguration to the manager since it is just the coordinator of the communication among the different computational units. In order to implement the two kernels with partial reconfiguration on the same reconfigurable area it is fundamental that they expose the same interface to the system, which in this case means that they should receive the same number of inputs[1]. For example, let us assume that we would like to implement the two kernels shown in Figure 4 by exploiting the partial reconfiguration. This solution can be easily implemented noticing that we can introduce a second parameter in the first kernel to have the same interface as the other one, without changing its functionality. Moreover, it

[1]The solution of unifying/extending kernel IOs until they match is quite simple but this step is necessary for the integration into MaxCompiler/Managers. We are working to have it done automatically.

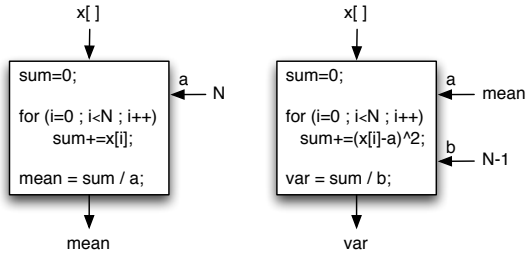

Fig. 4. Static implementation of two kernels computing the mean and the variance of the input dataset.

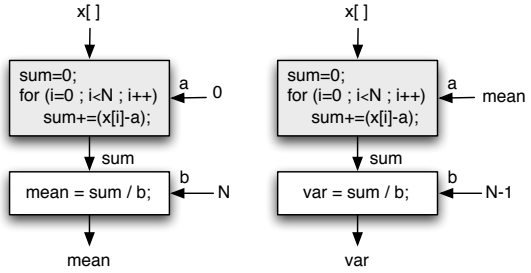

Fig. 5. Definition of the kernels to allow to change the functionality only by reconfiguring the gray parts of the kernels and sharing the second one.

is not necessary to reconfigure a smaller portion of a single kernel and thus kernels have been intended also as the atomic reconfigurable part of the architecture. In fact, if someone needs to slightly modify the behavior of a kernel (*e.g* dividing something by three instead than by five), this can be obtained by parameterizing the kernel. Furthermore, if the modification is very small, but with differences from a logical point of view (*e.g.* substituting $a + b$ with $a - b$ in the last step of the computation), it is possible to break-up the design into smaller kernels to perform the same functionality.

In the classical Maxeler design flow, the entire compilation flow starts invoking the `build` method in the Manager, which translates the MaxJ code into VHDL, invokes synthesis tools and produces the `.max` file to configure the FPGA. Moreover the manager is responsible for the kernel instantiation and the definition of the streams connecting the kernels and the host application. To be able to change the design flow, we decided to implement a new type of manager to support PR at this level, where we can represent the reconfigurable regions, the interconnections among them and their configurations.

Then, since the standard Xilinx PR design flow requires a user to define area constraints and placement for each reconfigurable region that need to be used, we decided to introduce a level of abstraction to represent the reconfigurable areas in MaxJ. In this way, the designer can decide in which part of the FPGA to place the reconfigurable blocks through a simple MaxJ function. Note that the automatic placement of these blocks is out of the scope of this work. A possible future work is to hide the process of area definition of these partitions to the user, by automating it inside the MaxCompiler. In fact, automating the area constraints definition would require the presence of some tools (e.g., [7]) able to optimize the choice of the area on the base of the Kernel characteristics.

Finally, the last step of the proposed approach has to establish how to physically reconfigure the device. As shown

in Section II, the IFPGA contains a VHDL state machine that controls the *CFPGA* configuration, carried out through the SelectMAP. In particular, the bitstream, contained inside the `.max` file, is loaded from the host and passed through the *IFPGA*, and this VHDL state machine is responsible for its loading into the *CFPGA*. Thus, the possibly simplest solution was to replicate a similar behavior, extending it with the support for the partial reconfiguration.

The result of the proposed approach is that, from the user's point of view, supporting the PR in the Maxeler flow does not require any big change in the standard Maxeler design: implementing a partially reconfigurable design can be considered very similar to the implementation of a static design and the designer would not notice any difference in the Maxeler toolflow, apart from the modifications in the definition of the reconfigurable blocks.

B. MaxCompiler Modifications

Different modifications have been introduced also into MaxCompiler in order to support the PR: in particular, they affect both the hardware side (the MaxJ code of the DFE, i.e., kernels and manager) and the software one (the *SLiC* interface of the host code) of the application.

Concerning the design of the DFE blocks, MaxCompiler libraries need to be modified to allow the creation of Reconfigurable Partitions and the support of a different design flow needed to generate a partially reconfigurable design. In particular, as described above, we introduced the *Partial Reconfiguration Manager* class (`PRManager`) in order to have the structure of a partially reconfigurable design similar to a standard one. Inside a `PRManager`, to define a Reconfigurable Area, a `ReconfigurableBlock` must be instantiated. Its constructor takes as input the coordinates of the clock regions assigned to that reconfigurable area and the list of the kernels have been assigned to this region. This means the one reconfigurable block can occupy one or more clock regions, and not a fraction of a single clock region. For this reason, the maximum number of reconfigurable regions that can be created correspond to the number of clock regions (e.g., 18 in the MAX3 DFE board). Note that this is not a limitation since, in the future, we plan to extend the workflow to specify the coordinates of the regions in a different and more customized way. During the FPGA execution, there will be exactly one kernel per reconfigurable block at the same time: the kernels specified as input to the `ReconfigurableBlock` constructor represent the possible configurations of the block. After these reconfigurable blocks have been instantiated in the `PRManager` and the corresponding kernels have been specified, the reconfigurable blocks can be used just like kernel blocks. In particular, the designer can define connections among the reconfigurable blocks, other kernels and the host code in the same way as in a regular `CustomManager`. Moreover, since the `PRManager` is a specialization of the `CustomManager`, it is possible to add one or more `KernelBlock` in the `PRManager` to place *static* kernels into the design.

After the `PRManager` has been instantiated, along with all the `ReconfigurableBlocks`, the designer must specify at least one `Configuration` for each of them. A `Configuration` represents the assignment of one and only one kernel to each reconfigurable block. In the subsequent steps of the design flow, a partial bitstream will be generated for each pair `Kernel-ReconfigurableBlock` of each `Configuration`. The pairs represents all the partial bitstreams that will be added to the `.max` file to allow the partial reconfiguration. Note that all possible unique `Configurations` could be enumerated and built automatically by MaxCompiler. This would simplify usage for the developer at the cost of increased build time.

Figure 6 summarizes the steps that the designer needs to adopt to create a partially reconfigurable design in the proposed extension of the Maxeler toolchain: the creation of the `PRManager`, the specification and interconnection of the `ReconfigurableBlocks`, the definition of the `Configurations` and, finally, the actual build and synthesis of the system.

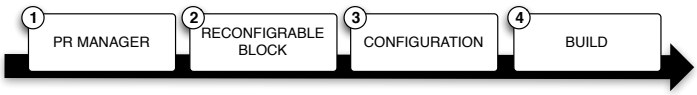

Fig. 6. DFE flow from the designer's perspective.

On the other side, concerning the development of the host code, some features must be added to the *SLiC* interface in order to issue the partial reconfiguration of the device. Indeed, since the concept of *PR* intrinsically implies a *dynamic* management of the device, the only way for the user to modify the *DFE* configuration at run-time is to invoke some *SLiC* functions from the host code to interact with the *CFPGA* and execute the partial reconfiguration of the *DFE*. For this reason, in order to exploit the PR directly from the application, we added the function `max_reconfig_partial_bitstream(device, kernel_name)` to the *SLiC* interface. By calling this function, the designer can reconfigure the reconfigurable block which the kernel identified by `kernel_name` belongs to by loading the corresponding partial bitstream. Note that it is not necessary to supply the `ReconfigurableBlock` as kernel names must be unique in MaxCompiler and a given instance of Kernel cannot be used in multiple `ReconfigurableBlocks`.

IV. Design Flow Implementation

The standard MaxCompiler flow is basically *linear*: starting from the source code, it produces a `maxfile` using all the needed tools (like Xilinx `ngbuild`, `map`, `par` and `bitgen`) once in sequence. When implementing a reconfigurable architecture, instead, the design flow becomes in some way *parallel*. In fact, in a PR design, the first step requires the implementation of the static part of the system, while the second step refers to the implementation of the desired configurations for the reconfigurable areas. Basically, once the static part of the system have been specified and synthesized all the configurations can be processed in a parallel manner. Moreover, PR requires to specify each configuration (through

the use of the PXML file [8]) and to modify the UCF file adding the area constraints which limits the mapping of the reconfigurable areas. The MaxJ class managing the entire partial reconfiguration process is the `PRManager`, which is the class where most of the MaxCompiler modifications have been implemented.

The implementation of the Xilinx PR flow into the Maxeler infrastructure has been realized as a three stages flow as illustrated in figure 7. Section IV-A explains each one of these stages, while Section IV-B provides details about how the partial reconfiguration is performed on the architecture, also describing modifications introduced in the SLiC interface.

A. Implemented Flow

*1) **First Stage**:* When the designer starts the build process, the first stage is a checking phase, where the modified version of the MaxCompiler checks inputs and outputs of the kernels (scalar and vectorial, memory mapping, etc...) inside the same `ReconfigurableBlock` to control if they are compatible. Furthermore, the MaxCompiler also checks that a single kernel instance is not part of two different `ReconfigurableBlocks`, because this would lead to name clashes when calling the function. Finally, there is a check to ensure that all the `ReconfigurableBlocks` have at least one kernel assigned and avoid that more than one kernel is assigned to a `ReconfigurableBlock` in the same `Configuration`.

After the checking phase, the translation from the MaxJ code to the VHDL representation of kernels and manager takes place. Note that the reconfigurable regions are treated as black boxes with the same inputs and outputs as the kernels that will be reconfigured dynamically into the black box.

When the VHDL code has been produced, `xst` and `ngcbuild` are called: the static part and the kernels in the `ReconfigurableBlocks` are synthesized independently from each other, producing all the needed netlists.

*2) **Second Stage**:* Within the second phase, all the PXML [8] files (one per configuration) are automatically generated to be passed to the proper synthesis tools. Moreover, the constraints of the `ReconfigurableBlocks`, specified by the developer when instantiating them, are included into the UCF file.

The next step consists in calling, for each configuration `mppr`, `pr_verify` and `bitgen`. `mppr` (multiple place and route) runs `map` and `par` multiple times in parallel to find the best placement. The `pr_verify` [8] utility is used to compare routed NCD files from two or more configurations created for a PR design to validate that all the imported resources match. For this reason, it is called for every `Configuration` but the first one, because when the first `Configuration` is built, there is not any other configuration to compare to.
Resources that need validation include:

- **Global Clock Spines**: Each global clock must have clock spines routed within the same clock regions in all the configurations;
- **Regional Clock Spines**: Each regional clock must have clock spines routed within the same clock regions in all the configurations;
- **Proxy logic**: Proxy logic, although logically part of the static design, must be placed at the same locations within the Area Groups allocated for the reconfigurable partitions;
- **Partition Interfaces**: Each reconfigurable partition must have the same input and output ports in each configuration.

If the `pr_verify` utility does not fail, it means that it is possible to import the results from previous configurations in the current one. The last Xilinx tool that is invoked is `bitgen`, which for each `Configuration` generates a full bitstream, and one partial bitstream per kernel in the `Configuration`. The full bitstream contains the static part of the design as well as all of the kernels specified in the `Configuration`.

*3) **Third Stage**:* The final stage of the building process consists in creating the `.max` file and compiling the CPU code. The CPU code compilation does not require any modification (it consists simply of C/C++ or FORTRAN code), while the `.max` file needs to include the specification of the partially reconfigurable design. The `.max` file for a static design contains an array storing the full bitstream. This bitstream is loaded into the DFE as soon as the application starts.

In case of a design containing partially reconfigurable areas, there is an array storing the full bitstream of the *default* `Configuration` plus as many arrays as the number of reconfigurable kernels in the design, containing their partial bitstreams. Obviously, even if one kernel is used in more than one configuration, the `.max` fill will contain just one partial bitstream to be loaded multiple times. The partial bitstreams have the same name as the kernels they are describing. For this reason, in the host code, it is sufficient to specify the name of the kernel that the designer would like to execute for issuing the reconfiguration. This loads its partial bitstream onto the FPGA; in detail, when the function `max_reconfig_partial_bitstream` is invoked, the selected partial bitstream file is taken from the `.max` file and loaded onto the FPGA.

B. Partial Reconfiguration Process

The function `max_reconfig_partial_bitstream` that has been described above is implemented in C in the *SLiC* interface and its insertion required also to modify the *IFPGA* (described in Section II). The IFPGA is configured with a full bitstream, stored in a flash memory, when the device is turned on. In the IFPGA, a VHDL state-machine is implemented [9] [10] to guide the configuration of the CFPGA: when it receives the command for configuring the CFPGA, it starts loading the full bitstream and manages the occurrence of errors.

To support PR, it has been added a new command to load the partial bitstreams. It works in a slightly different way than the command for the full configuration, because of the vendor-specific distinctions between a full and a partial reconfigurations (e.g. the `DONE` signal is not asserted). Thus, when calling the function `max_reconfig_partial_bitstream` from

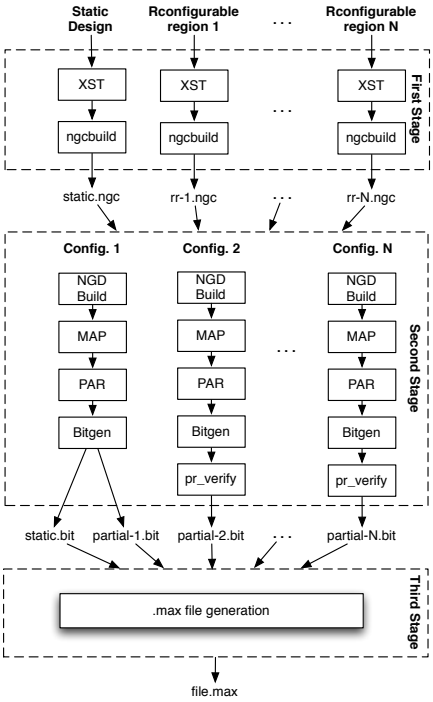

Fig. 7. Maxeler flow with support to partial reconfiguration

the host code, a command is passed to the VHDL state-machine in the IFPGA to load the selected partial bitstream. The IFPGA then reports if either the partial bitstream has been loaded with success or an error has occurred. Since the full configuration of the CFPGA is carried out by the VHDL state machine in the IFPGA through the `SelectMAP` [11] configuration interface, the PR makes use of the `SelectMAP` as well.

V. Evaluation Results

We evaluated the introduction of the partial reconfiguration on the Maxeler architecture by analyzing a case study that, even in its simplicity, captures all the interesting benefits. The platform used for testing is a Maxeler MaxWorstation [6] which is equipped with an Intel Core i7 (quad-core) with 16GB of RAM and coupled with a Virtex-6 XC6VSX475T FPGA (476,160 logic cells) with 24GB of private and dynamic RAM. The FPGA is connected to the CPU through a PCI bus granting a maximum throughput of 2GB/s and allowing a parallel transfer of, at most, eight streams of data. The application used in the test are the Canny Edge detector [12], further referred as CANNY, which is an image processing filter and thus is well suited for hardware implementation and is usually used as test application [13]. In the rest of this section, we describe the analyzed application (Section V-A) and the resulting architectures (Section V-B) generated exploiting the Maxeler framework enhanced with PR capability. Videos showing the implemented demos can be seen at the EU FASTER project YouTube channel (https://www.youtube.com/user/FASTERprj). Finally, Section V-C contains the analysis of the obtained results.

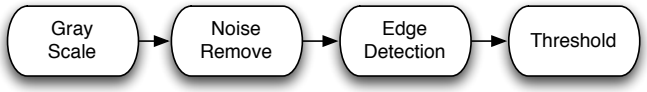

Fig. 8. Dataflow representation of the Canny algorithm.

A. Canny Edge Detector

The edge detection is an image filtering process aimed to identify edges in an image analyzing color transition between adjacent pixels. This process is carried out computing the discrete approximation of the 2D derivative (derivative along horizontal and vertical axis) of the color intensity of a pixel. As many other image processing algorithms, the edge detection filter operates on a gray scale image as input since most of the relevant information are captured by the color intensity itself. Given the input image, the 2D derivative discrete approximation is given by the convolution of the image with the matrix of the filter. In our implementation, we decided to use the Laplacian operator to compute the derivative which is characterized by the following matrix:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The result of the convolution is highly influenced by the noise in the image and that leads to poor edges identification. The idea of the CANNY filter is to pre-process the image with a noise removal filter and then apply a threshold after the edge detection filter to remove false positives. We implemented the noise removal phase using another convolution filter, which is characterized by the following matrix:

$$A = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The whole filter is thus composed of 4 steps, as shown in Figure 8, performed in the following order: (1) Gray Scale conversion, (2) Noise Removal filter, (3) Edge Detection filter, (4) Threshold phase.

B. Architectures

Four different partial reconfigurable architectures have been generated within the new version of the Maxeler flow. In these experiments the CANNY pipeline has been divided with two reconfigurable regions: Gray Scale (GS) and Edge Detection (ED) have been implemented in the first region (RR1), while the Noise Removal (NR) and the Threshold (TH) phase have implemented in the second one (RR2); this configuration is reported in Figure 9. The different implemented

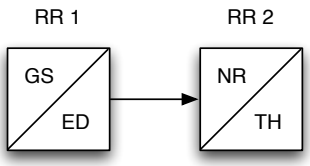

Fig. 9. Dataflow representation of the Canny algorithm pipeline when implemented in a reconfigurable design.

architectures expose a different level of parallelism for the analyzed application, from one to four parallel kernels for each phase. A representation of these architectures is reported in Figure 10. This would allow to analyze the trade-off between performance and requirements of resources. We analyzed the behavior of these architectures by applying the filter to 1,462 frames of an input video file, where each of them has a resolution of 1024×768.

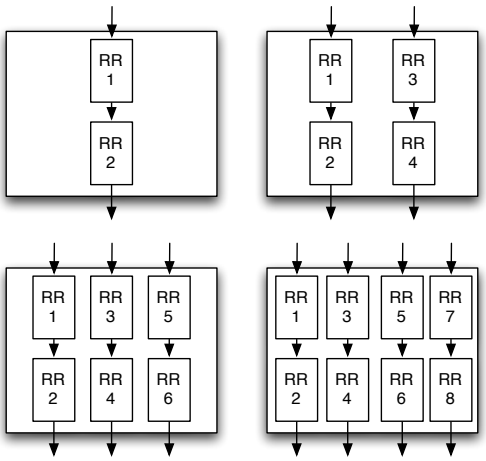

Fig. 10. Implemented architectures with a different number of pipelines.

C. Analysis of the Results

First, we statically implemented the pipeline of the CANNY reported in Figure 8. This implementation, without using Partial Reconfiguration, occupies $680+681+632+500 = 2,493$ LUTs (considering the area occupation of each kernel) for each of the pipeline. The largest architecture (obtained with four replications) occupies $6,320$ LUTs for the communication logic and, thus, $6,320 + 2,493 \times 4 = 16,292$ LUTs for the entire system. Table I reports the results of the tests that we performed by implementing the application described above in the enhanced version of the Maxeler toolchain proposed in this paper. The table shows the area occupation, the execution

TABLE I
DETAILS OF THE IMPLEMENTED ARCHITECTURES

Pipelines	Reconf. Regions	Area [LUTs]	Execution Time [s]	Reconf. Time [s]	Throughput [fps]
1	2	6664	72.845	0.097	20.06
2	4	8326	40.953	0.219	35.69
3	6	10117	31.125	0.320	46.97
4	8	11764	26.063	0.400	56.09

and reconfiguration time and the overall throughput for each the evaluated architectures, characterized by the number of the pipelines. As expected, the reported values scales linearly with the number of available pipelines, ranging to a minimum performance of 20 frames per seconds to the fastest architecture that is able to process up to 56 frames per seconds. Concerning the area occupation, each pipeline in the reconfigurable architecture (as shown in Figure 9) occupies $1,361$ LUTs, equally balanced between the 2 PRRs. Considering the last implementation (with 4 pipelines) we have that the static design, implementing basically the communication logic between the regions and with the host, occupies $6,320$ LUTs, while the 4 pipelines require $1,362 \times 4 = 5,444$ LUTs. Then, the entire system occupies $6,320 + 5,444 = 11,764$ LUTs; as a result, the PR implementation allows in this case to reduce the area occupation by 28%. In conclusion, the generated architectures allow to obtain a good result in terms of performance (elaborating more than 56 frames per second), while reducing the area with respect to the static implementation. This would allow to fit more complex applications on the same device, but also to deploy multiple applications without violating the area constraints. To conclude this case study, it is worth noting that, using the Maxeler flow, all the hardware is programmed in a Java-like language. This has the great advantage that the designer can simply describe the kernels with a high-level language, without taking care of the hardware details. Moreover, he or she can also compose the higher level structure of the application (i.e., the manager) by adopting a Java-like language as well (with an input parameter, that is the number of desired pipelines). As a result, we have been able to design all the four architectures simply starting from the same source code of the single kernels and then changing only one input parameter. At the best of our knowledge there are no tools that can provide this level of abstraction for the design of reconfigurable architectures, especially for dataflow computation. An interested reader can found the original video an the computed output at [14].

VI. Related Work

Since the advent of reconfigurable computing many efforts have been made to incorporate reconfigurable elements and a traditional processor within a single chip. The aim of designing such hybrid chips, is the acceleration of computationally intensive tasks in the reconfigurable array that otherwise would execute in the software processor. In many application domains, code sections such as loop statements can be efficiently deployed in the reconfigurable array, and the application can benefit significantly from the inherent parallelism and pipelining the reconfigurable hardware offers. Early works towards this direction are PRISM developed at Brown University, and the GARP project from UC Berkeley which was envisioned to fit into an ordinary processing environment with structured programs, libraries, context switches, virtual memory, and multiple users. Today, several devices combine software processors and reconfigurable hardware within the same silicon. Also, many systems use a processor to control the FPGA reconfiguration, either with single software routines or through an operating system, while other proposals relieve the processor using a dedicated reconfiguration controller. The latest Xilinx Virtex-7 devices contain ARM cores which are connected to fine-grained reconfigurable hardware resources through a high-performance channel architecture that supports memory-mapped transactions and a streaming interface. Within such a context, Maxeler platforms have been demonstrated to outperform *standard* solutions in fields related to hardware acceleration and dataflow computing. In [15], authors defined and explored various metrics to describe

the design and performance trade-offs of modern commercial computer architectures with particular emphasis on hardware acceleration solutions based on Maxeler's DFE and CUDA-enabled GPUs. They compare the MAX2 board, the Tesla C1060 GPU and the Intel Nehalem E5520 performance, concluding that each architecture has its own advantages and disadvantages. FPGAs, in particular, leverage on their high flexibility and customizability to offer the best trade-off between peak performance and energy consumption. In [16], the authors make a similar comparison (CPU, FPGA and GPU) and conclude that the Maxeler FPGA-based heterogeneous system is the most power-efficient, even though not necessarily the architecture yielding the highest peak performance.

Authors agree that for applications relying on very large datasets and complex numerical computations, the Maxeler dataflow machine offers competitive or better performance with respect to other data flow-oriented platforms and architectures [15]. In [17] Maxeler platforms are used to compute 3D finite difference modeling, a compute-intense numerical method applied to different disciplines, such as geosciences, medical imaging and physics simulations. In this article, finite difference is used to model the wave propagation through earth. User program is mapped directly into hardware by using a programming environment called *MaxGenFD*. MaxGenFD is a domain specific compiler for 3D finite difference applications designed to enable geoscientists and programmers to effectively harness the compute power of dataflow hardware without learning circuit design. It is specifically targeted at implementing seismic wave forward modeling, namely *RTM* and *FWI*. Experimental results showed relevant speedups with respect to GPU-based and CPU-based solutions, with lower overall power consumption [18].

The first work which tried, in some way, to combine PR with Maxeler platforms is the one presented in [19]. The aim of the work is to exploit partial dynamic reconfiguration in a system based on the interaction between a FPGA and a CPU to accelerate databases management system, with the work of an FPGA based query processor with a regular software database. Despite the not completely successful outcome, the work in [19] shows some interest for Maxeler platforms and their possible usage for accelerating data filtering queries; for this type of work the possibility to exploit PR in the MaxCompiler flow would have been greatly beneficial, because it would have allowed the author to easily create the partially reconfigurable modules and to focus just on accelerating the databases management system.

VII. Conclusions and Future Work

The work presented in this paper enhances the potentialities of Maxeler platforms introducing the concept of Partial Reconfiguration. Maxeler systems are designed to increase dramatically the performance of a given application. Partial Reconfiguration allows to build more flexible systems, able to modify their behavior at run-time to *adapt* to the environment and to user's needs. Besides improvements in the application's raw speed (obviously, depending on the application itself), a partially reconfigurable design can mitigate the problems related to FPGA capacity constraints that can be encountered in the classical static designs. The presented work has proven to be effective and successful, as shown in Section V with the video filtering application. However, the work has still different possibilities of improvement, e.g., the instantiation of a `ReconfigurableBlock`. In fact, the decisions about which portion of the FPGA must be reserved to a given `ReconfigurableBlock` is the only part that is not fully automated in the MaxCompiler flow when using Partial Reconfiguration.

Acknowledgments

This work was partially funded by the European Commission in the context of the FP7 FASTER project [20](#287804).

References

[1] "What is an FPGA," Altera Corporation. [Online]. Available: http://www.altera.com/products/fpga.html
[2] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, "Features, Design Tools, and Application Domains of FPGAs," *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 4, pp. 1810–1823, 2007.
[3] "Cray Inc." [Online]. Available: http://www.cray.com/Home.aspx
[4] I. Corporation, "Intel embedded website: Intel atom processor E6x5C series-based platform." [Online]. Available: http://www.intel.com/p/en_US/embedded/hwsw/hardware/atom-e6x5c/overview
[5] "Maxeler Technologies Ltd. website." [Online]. Available: http://www.maxeler.com
[6] "Maxworkstation," Maxeler Technologies Ltd. [Online]. Available: http://www.maxeler.com/products/desktop/
[7] A. Bonetto, A. Cazzaniga, G. Durelli, C. Pilato, D. Sciuto, and M. D. Santambrogio, "An open-source design and validation platform for reconfigurable systems," in *22nd International Conference on Field Programmable Logic and Applications*, 2012.
[8] Xilinx Inc., *Partial Reconfiguration User Guide*, Xilinx Inc., Jul 2012.
[9] D. L. Perry, *VHDL: Programming by Example*, 4th ed. New York, NY, USA: McGraw-Hill, Inc., 2002. [Online]. Available: http://www.iitg.ernet.in/asahu/cs223/VHDL.Programming.DouglasPerry.pdf
[10] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Orlando, FL, USA: Academic Press, Inc., 2004.
[11] "Virtex-6 fpga configuration," Sep 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf
[12] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 679–698, Jun. 1986.
[13] C. Pilato, A. Cazzaniga, G. Durelli, A. Otero, D. Sciuto, and M. D. Santambrogio, "On the automatic integration of hardware accelerators into FPGA-based embedded systems," in *22nd International Conference on Field Programmable Logic and Applications*, 2012, pp. 607–610.
[14] Video sample. [Online]. Available: http://tinyurl.com/raw2013-maxPR
[15] S. Stojanovic, D. Bojic, M. Bojovic, M. Valero, and V. Milutinovic, "An overview of selected hybrid and reconfigurable architectures," in *2012 IEEE International Conference on Industrial Technology*, ser. ICIT '12, Mar 2012, pp. 444–449.
[16] Q. Liu and W. Luk, "Heterogeneous Systems for Energy Efficient," in *international conference on Reconfigurable Computing: architectures, tools and applications (ARC)*, 2012, pp. 64–75.
[17] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. J. Flynn, "Finite difference wave propagation modeling on special purpose dataflow machines," *IEEE Transactions on Parallel and Distributed Systems*, June 2012. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/TPDS.2012.198
[18] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications," *IEEE Micro*, pp. 41–49, 2011.
[19] J. J. Jensen, "Reconfigurable FPGA accelerator for databases," Master's thesis, University of Oslo, Aug 2012.
[20] Faster website. [Online]. Available: http://www.fp7-faster.eu/