Migrating Static Systems to Partially Reconfigurable Systems on Spartan-6 FPGAs

Christian Beckhoff
Email: christian@recobus.de

Dirk Koch and Jim Torresen
Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
Email: {koch, jimtoer}@ifi.uio.no
Web: http://www.matnat.uio.no/forskning/prosjekter/crc/

***Abstract*—In this paper we present a design flow for migrating a static only FPGA system into a system featuring partial runtime reconfiguration. The migration can lower device cost and benefits future extendibility. The migration flow with our tool GoAhead is described in detail. In contrast to present EDA tools, GoAhead is the first tool that allows to build systems with partial runtime reconfiguration on Spartan-6 FPGAs. A case study will demonstrate the migration of a static only design into a partially reconfigurable system.**

I. Introduction

With the Virtex-6 family, Xilinx offers FPGAs with high logic density targeting high-performance applications. Compared to Virtex-6, the cheaper Spartan-6 FPGAs in turn provide a lower logic density and consume less power than Virtex-6 FPGAs.

While there are tools available for partial reconfiguration of the more expensive Virtex-6 FPGAs, there is no such Xilinx tool support for partial reconfiguration of the cheaper and less power consuming Spartan-6 devices. Hence, today a user aiming to save device costs with partial reconfiguration is bound to Virtex-6 FPGAs. Although, partial reconfiguration on Virtex-6 FPGAs lowers the device cost, the reduced power and money expenses may not reach what is achievable by a Spartan-6 FPGA with partial reconfiguration. Similarly a static only implementation on a Spartan-6 FPGA might outperform a reconfigurable Virtex-6 implementation in both unit cost and power consumption.

In other words: A user has to start with an expensive device in order to save money with partial reconfiguration. This dilemma is caused by the lack of tool support for partial reconfiguration on Spartan-6.

Concepts for partial reconfiguration on less recent devices like Spartan-3 and Virtex-2 are subject of various publications [1], [2], [3], [4]. However, less research on partial reconfiguration for Spartan-6 FPGAs is available [5]. In this paper, we present our novel tool *GoAhead*. GoAhead closes the tool gap and enables partial reconfiguration for Spartan-6 FPGAs and thus provides an option to lower device costs. Hence, using a Spartan-6 FPGA might even be more efficient than an ASIC solution for certain applications where many functional blocks are execute mutually exclusive.

Our tool GoAhead will replace our previous tool ReCoBus-Builder [2] by further improving usability (e.g. by a complete scripting interfaces and a wizard for building a system with partial reconfiguration). GoAhead supports the recent Virtex-6 and Spartan-6 FPGAs. We also include new features into the tool such as the capability to route static signals through a partial area using routing resources that will not be used by partial modules and consequently without interfering any partial module. Thus, we support module relocation also on Spartan-6 FPGAs.

The overall flow with GoAhead is presented in Figure 1. The flow allows to build a reconfigurable systems from scratch or to migrate an existing static only design to a reconfigurable design. When following our flow, the static system can be implemented completely independent from the partial modules. This lowers the total synthesis duration especially for design spins. In addition, after changes in the static system, none of the partial module will have to be reimplemented again, as long as the interface between the static and the partial system remains unchanged.

The design migration step is outlined in Section II. The migration step is succeeded by the physical implementation of both the static part and the partial modules. Both branches are carried out with the Xilinx tools constrained by special macros and placement information generated by our tool. The physical implementation of the static part is presented in Section III followed by the physical implementation of the partial modules in Section IV. In Section V we reveal a case study where we migrate a static system into a partial one. After this, in Section VI, we summarize the main differences between GoAhead and the Xilinx vendor tool PlanAhead.

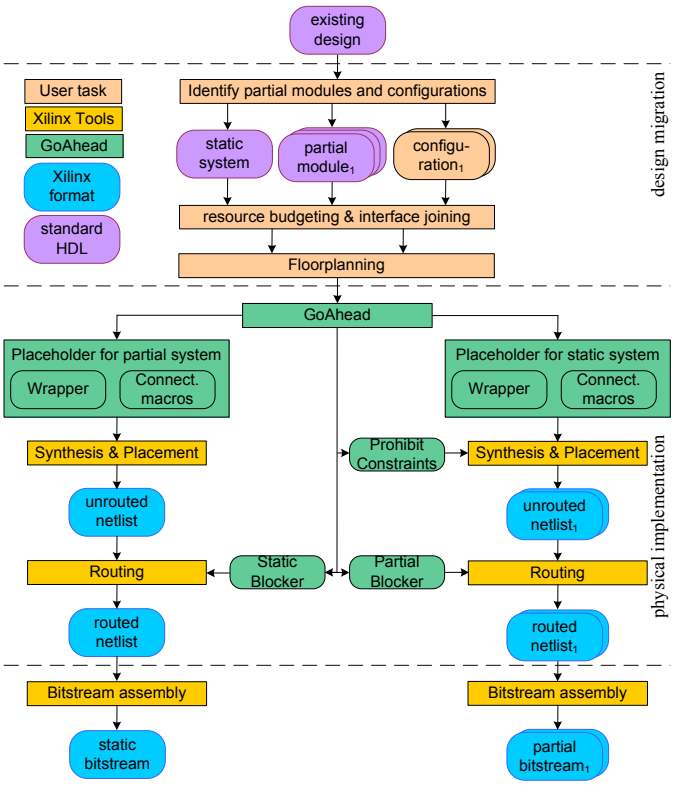

Fig. 1. The GoAhead migration flow allows to convert an existing static design into a partially reconfigurable system. The migration part requires user interaction. The two branches in the flow are completely independent of each other. Hence, the static system can be (re)built without rebuilding the partial system and vice versa.

II. Design Migration

A widely spread platitude tells us, that a running system should not be touched. However, there are several reasons to migrate an existing design into a design with partial runtime configuration.

It could be necessary for example to enhance an existing design with an additional module. If though, all resources on the FPGA are allocated already, the enhancement would require to replace it with a larger device. If the system is shipped already, the replacement would be very expensive. A system with partial reconfiguration on the other hand might prevent the need for a replacement, if existing modules and the new module can share resources exclusively over time. Thus the shipped hardware would remain unchanged, while only the FPGA bitstream and the driver software changes.

Prototyping might also be enhanced by partial reconfiguration. The hardware team could focus on implementing the static part of a system. If the static system is available, the software team could start to develop and test firmware for the static part of the system, while the hardware team then implements the partial modules. Thus, concurrent development of hardware and software would be increased, and contributing speeding up the overall design cycle.

In the following, we will outline how a static system can be migrated to a system with partial reconfiguration composed of a static part and a partial area hosting modules over time.

A. Identifying Partial Configurations

For migrating a static design towards a system featuring partial reconfiguration, we have to examine the existing design and extract those submodules which can share resources exclusively on the FPGA fabric and those which remain in the existing design static only.

Partial modules might be identified by multiplexers that are used to switch between functional blocks that are possible candidates to be shared in one reconfigurable area. Thus, instead of switching the multiplexer for selecting a mutual exclusion function, partial reconfiguration can be used for this purpose. However, while in most cases the multiplexer can be switched within a single clock cycle, the reconfiguration takes time in the range of many thousands to millions of clock cycles, depending on the modules size (which correlates with the bitstream size).

Consequently, reconfiguration is bound to a relatively slow changing of functional units (e.g. for adapting to different operation modes). However, by applying techniques for high speed configuration [6] even complex modules might be exchanged fast (e.g. a Microblaze softcore CPU below in one millisecond).

For the following, let us assume a static only system as depicted in Figure 2 a). There are three possible data paths from the output of the Camera to the input of the DVI encoder:

1) Camera → Gaussian filter → Segmentation → DVI
2) Camera → Median filter → Segmentation → DVI
3) Testpattern → DVI

While the Camera and the DVI encoder are static only modules (as they are connecting peripherals), the two filters as well as the Segmentation and the Testpattern module can share the same FPGA resources over time. Hence, we identified the Gaussian filter gf, the Median Filter mf, the Segmentation module $segm$ and the Testpattern module tp as four partial modules. However, both gf and seg as well as mf and seg always run in parallel (i.e. at the same time). Only tp runs standalone. We will call modules that run at the same time a *partial configuration*.

For the three input data paths, we can derive one partial configuration each:

$$C = \Big\{ \{gf, seg\}, \{mf, seg\}, \{tp\} \Big\}$$

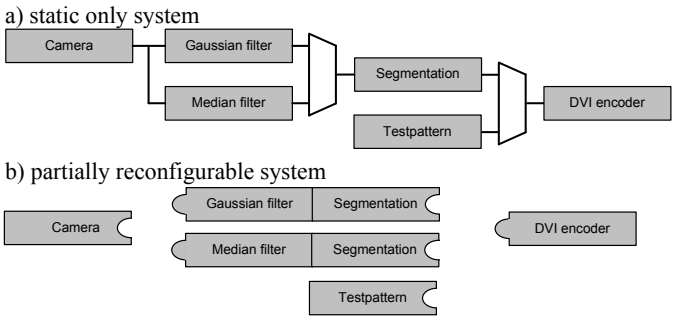

Fig. 2. a) Static only system consisting of a camera whose input is filtered and segmented and sent to a DVI encoder. Alternatively, a testpattern can be applied to the DVI encoder. In total, three input data paths for the DVI encoder exist within the system. b) The static only system divided into camera and DVI encoder along with three partial configurations of filters and testpattern. The Segmentation and the DVI encoder remain static. The three partial configurations share resources over time.

B. Resource Budgeting, Interface Joining and Floorplanning

Now, in order to determine the size of the reconfigurable areas that will host the partial modules, we have to identify the resource consumption of each partial configuration c_i. The resource consumption for a partial configuration c_i is a quintuplet $q_i = (\#LUT, \#BRAM, \#DSP, \#Input, \#Output)$, where $\#LUT$ is the number of look-up tables[1]. $\#BRAM$ and $\#DSP$ denote the number of required RAM blocks and DSP blocks, respectively the input and output width of each module is denoted with $\#Input$ and $\#Output$ and is specified in terms of single bit signals within the top-level entity of the different partial configurations $c_i \in C$. The parameters $\#Input$ and $\#Output$ can directly be identified from the module interfaces. Let us assume, the modules provide the interfaces given in Listing 1. Then, the resulting joined wrapper is given as listed in Listing 2.

Listing 1. Module interfaces of the reconfigurable modules in the example system of Figure 2

```
entity gaussian_filter is port
(
    video_in    : in std_logic_vector(...);
    video_out   : out std_logic_vector(...);
    filter_ctrl : in std_logic_vector(...)
);
end entity gaussian_filter;
entity median_filter is port
(
    video_in    : in std_logic_vector(...);
    video_out   : out std_logic_vector(...)
);
end entity median_filter;
```

[1]In practice, when implementing a reconfigurable system on Xilinx FPGAs, the logic consumption of a module will be specified in terms of slices, as a slice can neither be shared by two reconfigurable modules nor by a module and the rest of the system.

```
entity segmentation is port
(
    video_in    : in std_logic_vector(...);
    video_out   : out std_logic_vector(...);
    match       : out std_logic
);
end entity segmentation;
entity testpattern is port
(
    video_out   : out std_logic_vector(...)
);
end entity testpattern;
```

Listing 2. Joined wrapper of the module interfaces shown in Listing 1.

```
entity reconfigurable_area is port
(
    video_in    : in std_logic_vector(...);
    video_out   : out std_logic_vector(...);
    filter_ctrl : in std_logic_vector(...);
    match       : out std_logic
);
end entity reconfigurable_area;
```

The logic and memory consumption ($\#LUT$, $\#BRAM$, $\#DSP$) requires the user to synthesize each partial configuration once. After determining the requirement q_i for each $c_i \in C$, and for each resource type, we find the maximal value among all partial configurations and thus derive the minimal resource consumption of the partial area as $Q_i = (\overline{\#LUT}, \overline{\#BRAM}, \overline{\#DSP}, \overline{\#Input}, \overline{\#Output})$. In case, the system will contain several reconfigurable areas, this process has to be carried out for each singular set of partial configurations that corresponds to each reconfigurable area.

As an alternative for manual resource budgeting, our tool can extract a modules resource consumption out of the existing design automatically. This is achieved by analyzing the placed netlist of the existing design. Therefore, the existing design must be available as a netlist specification (i.e. an XDL-File [7], NCDs or NMCs can be converted to an XDL-File with a vendor tool). In the XDL file, the instantiation of slices (logic consumption), RAM blocks and DSP blocks is given in a human readable form (the routing is also available, but is currently not further considered). Each instantiation of a slice, RAM or DSP block contains the name of the module it belongs to. Hierarchies within the design are denoted by an arbitrary delimiter (e.g. a slash in $.../Testpattern/ramb16/...$). GoAhead is capable of finding common prefixes in netlist specifications which are used to automatically build the hierarchy; the tool then counts the resources used in the different levels of the hierarchy, which corresponds to the resource requirements of the modules. Thus, in addition to the manual synthesis of each partial configuration, we automate the resource budgeting. The automatic resource budgeting is part of our migration wizard

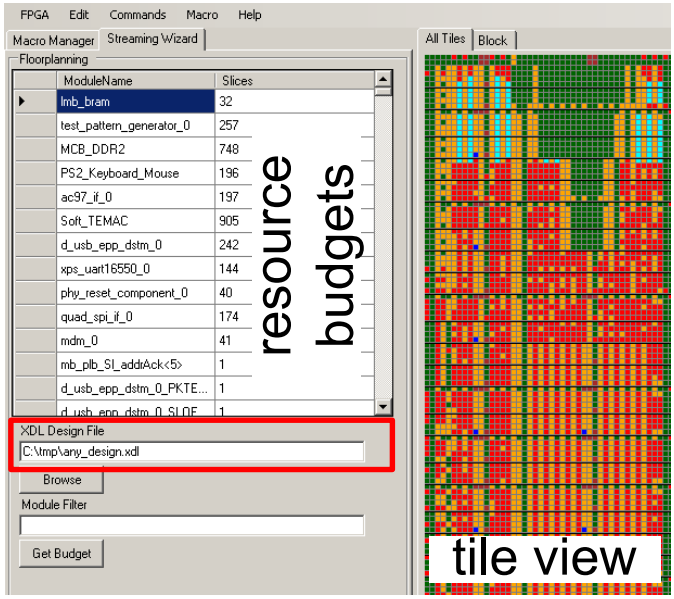

Fig. 3. Automatic resource budgeting in the GUI of our tool GoAhead. After reading in an existing design (XDL netlist), the used tiles are highlighted red in the tile view. For each detected module, the resource consumption (only slice are shown) is listed on the left.

and is depicted in Figure 3.

After budgeting the partial area, we will now define the partial area on the FPGA fabric. Here, the user can optionally assist in the definition of physical interfaces, which in particular comprises the assignment of signals crossing the boundary between the static part and reconfigurable area on an edge (left, right, top, bottom) of that area. In which direction a signal should cross the boundary of the static and the partial depends on the signals purpose. If, e.g. a memory controller is located left of the partial area, those signals of the physical interface that access the memory controller should be located on the left of the partial area as well. By manually defining the placement of each physical interface wire, the user can enhance the routing quality. In the following examples, we will assume that all signal enter the partial area on the left and exit the partial area on the right.

All signals of the physical interface use one individual PR link [8] to cross the boundary between the static part and the partial area. A PR link comprises a physical wire of the FPGA routing fabric as depicted in Figure 4. As in our approach, each partial module provides the same physical interface, the placement of the PR links is identical for all modules. However, the Xilinx tools provide no possibility to access a particular physical wire in HDL. Our tool will therefore generate blocker macros that block all routing resources in a given area except those wires we want the router to use as our PR links.

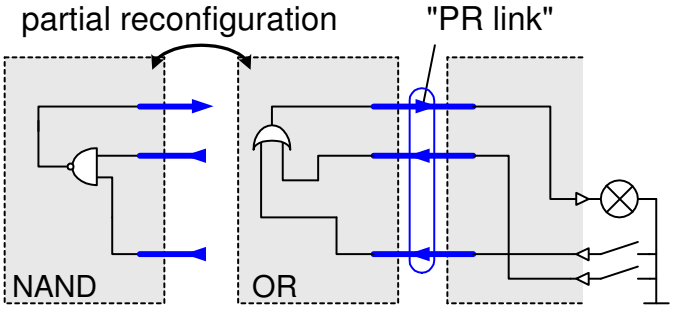

Fig. 4. A PR link [8] comprises a physical wire that crosses the boundary between the static part of the design and the partial area. All partial modules provide the same placement of PR links, hence we may swap modules over time.

III. GoAhead Static System Implementation

As introduced in Section I, it is beneficial to implement the static part of the system completely independent of the physical implementation of the different partial configurations. This removes, for example, the need to reroute all partial configurations if the static design changes.

In order to allow this strict encapsulation of the configurable modules, it requires the definition of bounding boxes for the different reconfigurable modules as well as a constraining of the routing between the static system and the partial modules. For the latter one, we have to bind each individual bit-signal of the partial configuration interface to a corresponding wire segment that physically crosses the border of the reconfigurable area. Unfortunately, the Xilinx vendor tools provide no corresponding constraints on the routing. However, as revealed later in this section, our tool is capable to generate such routing constraints.

When the static system is implemented, we will use a placeholder for the reconfigurable subsystem in order to ensure that no logic or routing will be trimmed. This is required as we want to implement the static system in absence of all partial modules (which, in some cases, might not even exist at compile time of the static system). During the implementation of each partial configuration, a placeholder for the static system will be used in turn. The placeholder is physically a set of hardmacros, called *connection macros* in the following.

As the first step after completing the resource budgeting and floorplanning, the tool places connection macros into the partial area. A connection macro is a simple hard macro that consists of one or more look-up tables without any routing. With each look-up table in the connection macro, we provide one input and one output port. The connection macros will be used to both connect a signal from the static part of the system with the partial area (module input) and to connect a signal from the partial area with a signal in the static system

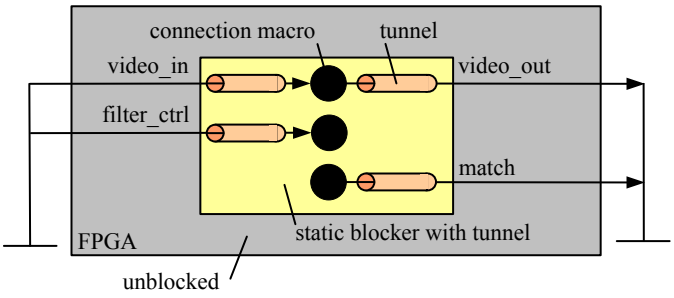

Fig. 5. A static blocker with tunnels. The reconfigurable area provides the joined wrapper for the reconfigurable area given in Listing 2.

(module output). We can connect a module input and a module output within one connection macro. The physical placement of the connection macros can be controlled by using Xilinx user constraints (UCF).

Our tool provides a wrapper for encapsulating the connection macros. Therefore, users can instantiate an automatically generated placeholder module which possesses the joined interface of the modules without being aware that the placeholder consists of hardmacros. This is the HDL-view of the implementation. During the physical implementation, the vendor placer will instantiate the macros according to the automatically generated placement constraints while later the router will perform the entire wire connection.

Note that, we do not connect signals from the static system with connection macros physically at this stage, but only connect static signals to connection macros in HDL. The physical routing will be done by the Xilinx router.

Although, our tool proposes placements for the connection macros, the user can define each placement in the GUI of the tool or via the scripting interface. Our tool then generates the placement constraints (UCF) for the connection macros that will be used during synthesis and placement.

The intermediate result after synthesis and the physical placement is an unrouted netlist. We now have to route this netlist and force the Xilinx router not to use any routing resource within the partial area. This is achieved by the help of *blocker macros* that can be generated with our tool.

The blocker macros occupy all routing resources in the partial area (static blocker), except the PR links we want the router explicitly to use to connect the static system and the placeholder macros. Consequently, the blocker macro blocks all routing resources, but leaves out a tunnel for each PR link. A static blocker macro with tunnels is outlined in Figure 5.

The router is then forced to use the tunnel to connect the assigned signal as this will be the only option within the blocker to reach the connection macro. By defining a tunnel in combination with the physical placement of a connection macro, we are able to bind a HDL signal to wire of the FPGA routing fabric. Though, not all wires of the Spartan-6 FPGA routing fabric are suitable for tunnels. A wire has to fulfill the following prerequisites:

1) Each end point of the wire can be extended to its corresponding begin point in order to extend the tunnel.
2) There may be no connection from an end point of a wire to the begin point of *another* wire that is used for another tunnel. This prevents the router from hopping between tunnels which would result in an inhomogeneous and unpredictable routing scheme.
3) Each end point of the wire can be connected to a look-up table input to connect the wire with the connection macro (module input).
4) A look-up table output from the connection macro can be connected to the begin point of the wire (module output).

The homogenous routing fabric of Virtex-II and Spartan-3 FPGAs offered plenty of wires fulfilling the above listed prerequisites. On Virtex-5 FPGAs, for most wires there are no direct connections from an end point of a wire to its corresponding begin point. Hence, stopovers inside and outside a tile are necessary. In the case of Spartan-6 and Virtex-6 FPGAs, Xilinx returned back to a more homogenous routing fabric but reduced the total number of wires which has to be considered if wide data paths have to be routed across the border of a reconfigurable area.

For Spartan-6 and Virtex-6 devices, the number of wires that can be interfaced in one CLB is bound to a value of four in practice. In GoAhead, we solved the problem of wide interfaces by optionally interleaving multiple wires that span a distance of multiple CLBs[2] and that start displaced in consecutive CLBs, as depicted in Figure 6. This approach is similar to the wide bus macros introduced temporarily by Xilinx for Virtex-4 devices [9].

The reduced number of routing resources in Virtex-5, Virtex-6 and Spartan-6 FPGAs facilitates blocker macros. For all three device families, it is possible to block all routing resource in a switch matrix of any tile by just using a few on tile look-up table outputs. As a consequence of the higher offer of routing resources in contrast to this simple blocking scheme, blocker macros on Virtex-2 and Spartan-3 were more complex. To block all routing resource within one particular tile assuming the presence of a communication architecture

[2]A CLB is a configurable logic block and comprises a cluster of in total eight 6-input look-up tables and an attached interconnect switch box for carrying out the FPGA routing. The CLBs are cascaded in a checkered manner together with dedicated resources such as memories or multiplier blocks

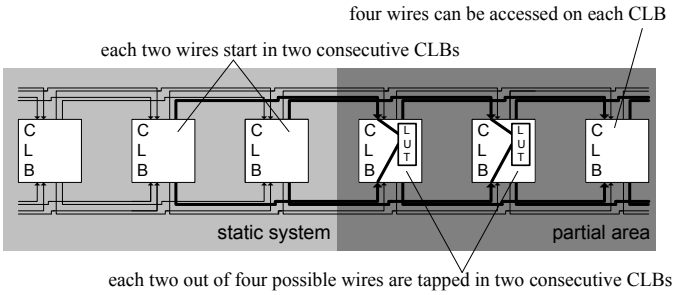

Fig. 6. A system comprising a static part and a partial area with a four bit interface. In two consecutive CLB in the static part only two wires out of four possible wires are connected. This results in a four bit interface interleaved into two disjoint tracks. A partial module needs again two consecutive CLBs to tap all four bits. A partial module that is only one column wide can only access two out of four interface bits.

such as a ReCoBus or an I/O-Bar [2] in this tile, one usually had to take advantage of drivers in other tiles.

After routing the static design with the help of connection macros and a corresponding blocker, we can finally generate a full bitstream with the Xilinx vendor tools. However, this static configuration will then not include any of the partial modules, but (after accomplishing the partial implementation) it is possible to generate an initial bitstream that includes partial modules directly after the initial reconfiguration [2].

IV. GOAHEAD PARTIAL MODULE IMPLEMENTATION

While the placeholder for the partial system was used to generate a bitstream for the static system, the placeholder for the static system in turn will be used to generate a bitstream for each module or for each partial configuration.

From the partial areas point of view the joined wrapper from Listing 2 will now appear as a reversed interface with each input becoming an output and vice versa. The partial module may not use any routing resources outside the partial area. Hence, a blocker macro (partial blocker) is surrounding the partial area (the macro will be generated by our tool). The partial blocker will spare out the same wire resources as the static blocker did building its tunnel.

To connect a static signal with an input of the partial area a connection macro will be placed inside the partial blocker that surrounds the partial area. The router now can only connect the connection macro with the module interface when using the before spared out tunnels. A partial blocker with connections macros is depicted in Figure 7). In addition, Xilinx placement constraints also generated by our tool will force the placer to place all module logic inside the partial area.

V. CASE STUDY

For demonstrating our migration flow, we implemented a static only system following the example shown in Figure 2a).

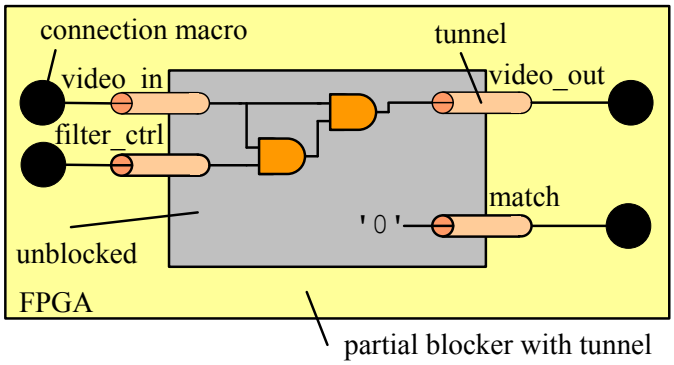

Fig. 7. Placeholder for the static system to implement the partial modules. The currently shown module is the Gaussian filter. A logical zero is driven to the unused output. The Median filter in turn would not access the input $filter\_ctrl$ but drive the output $match$. The testpattern module would access $video\_in$ and $video\_out$. All wires running into or out of the partial area run through tunnels (i.e. a set of unblocked wires).

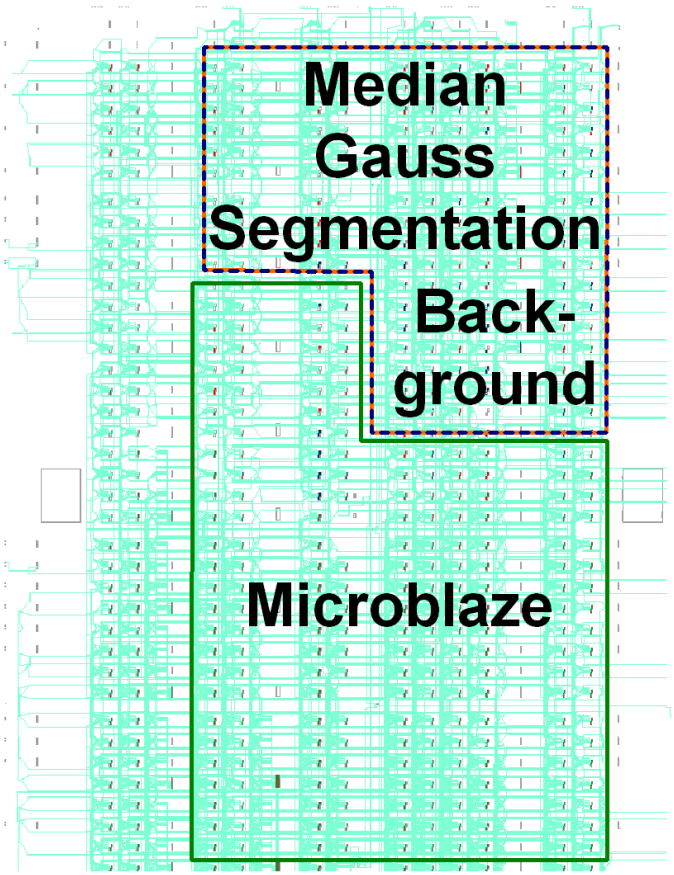

Fig. 8. FPGA-Editor screenshot of the initial static only system.

In addition to the depicted data path, the system has been equipped with a baseline Microblaze softcore processor for controlling the system. An FPGA-Editor screenshot of the static system is shown in Figure 8. The static only system occupies 94% of the available logic resources (slices) of the

TABLE I
RESOUCE REQUIREMENTS OF THE STATIC ONLY SYSTEM. THE TOTAL AMOUNT INCLUDES FURTHER MODULES AND GLUE LOGIC.

Module	# Slices	# BRAM	# DSP	# Input	# Output
Microblaze	814 (57%)	12 (37.5%)	3 (18%)		
Median_3x3	152 (10.6%)	4 (12.5%)	0	28	28
Gauss_5x5	161 (11.3%)	8 (25%)	0	29	28
Segmentation	64 (4.5%)	0	0	28	28
Background	110 (7.7%)	0	0	0	28
Total	1350 (94%)	24 (75%)	3 (18%)		

selected Xilinx Spartan-6 XC6SLX-9 device. We used the design analysis function provided by GoAhead to determine the exact resource requirements of the different modules in this system. The requirements are summarized in Table I.

It must be mentioned that the listed values for slices are not exact because a slice provides multiple look-up tables (like four in the case of Spartan-6) that might not by all used (different slice packing) or that might be shared among multiple different modules. However, the determined values have been found sufficient precise.

Table I lists three candidates (the median filter, the Gaussian filter, and the background module) that are suitable for partial reconfiguration, because only one of these modules is used at a point in time (depending on the present operation mode of the system). With the Gaussian filter being the largest module in terms of both logic resources and memory blocks, the partial area must be at least sufficient to host this module. Then, the maximal theoretical area saving would be the resources of the Median filter plus the resources of the background module, which is 216 slices (15%) and 4 BRAMs (12.5%). However, this assumes that the bounding box is defined as tight as possible, which might in some cases result in a poor routing. Furthermore, we have to add an interface for partial reconfiguration. In our case study, we used a simple PIO for this purpose that is directly connected to the 16-bit wide internal configuration access port (ICAP) of the Spartan-6 FPGA. Consequently, the CPU is capable to write configuration data for self-reconfiguring the device. Note that this is a slow but resource efficient option to interface a configuration port to a CPU.

The smallest reconfigurable unit of Spartan-6 FPGAs is a *frame* that contains a piece of configuration data for in total 16 CLBs or 32 slices. In the case of dedicated memory, one frame contains data for four BRAMs. When defining the reconfigurable region with two frames in height and 5 CLB columns in width, the reconfigurable region would provide sufficient resources for the Gaussian filter and the segmentation module (320 slices and 8 BRAMs). This is 49 slices more than the resource requirements of these two modules and the saving in logic resources would then be 216 - 49 = 167 slices or 11.7%

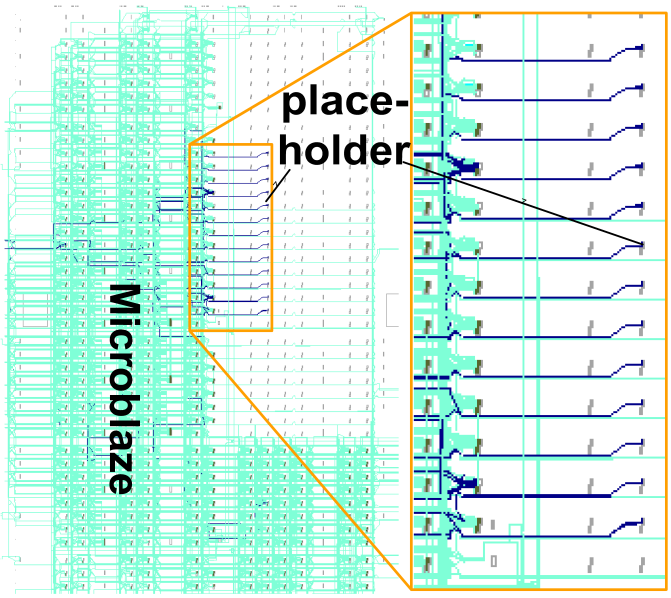

Fig. 9. FPGA-Editor screenshot of the modified static system providing a reconfigurable region.

of the total device resources. However, in the test system implementation, we maximized the reconfigurable region to 6 CLB columns in width which results in a reconfigurable region that provides 384 slices or 26.9% of the device, which is more appropriate for experiments on partial reconfiguration. The modified system which provides now a reconfigurable region to host the different video filters, the segmentation module and the background module is shown in Figure 9. Note that there are a few wires of the static system that cross the partial region. However, this is allowed and is in most systems required as Spartan-6 lack *long lines* that span the full height or width of a device. These routing resources are provided on older FPGAs, such as Xilinx Virtex-II devices, and are highly suitable to cross a partial region. GoAhead will generate constraints for the physical implementation of the modules such that the static routing will not interfere with the module routing.

The reconfigurable modules have been implemented following the same interface specification that has been used for the static system. Instead of a prohibit region, we used a bounding box definition for the physical implementation of the modules.

VI. COMPARISON OF GOAHEAD AND PLANAHEAD

In this section, we will compare our tool GoAhead with the competing Xilinx tool PlanAhead.

While our tool with Virtex-6 and Spartan-6 supports the most recent Xilinx FPGAs, PlanAhead does not support Spartan-6 so far. However, the low cost Spartan-6 family is a very adequate target for building systems with partial runtime reconfiguration. Our tool is the first to support Spartan-6.

In both tools, the floorplanning is carried out manually via the GUI, however scripting interfaces are available in PlanAhead and GoAhead.

To let wires cross the boundary between the static part of the system and the partial area, GoAhead uses the above mentioned connection macros. In PlanAhead on the other hand, this is accomplished with *proxy logic*. The proxy logic comprises one look-up table for each wire and is instantiated in the partial area. The module then taps the proxy logic. However, in the final module implementation, the proxy logic will still be present. In contrast, the connection macros in modules built following our flow are removed imposing zero overhead [8]. The placement of the proxy logic in PlanAhead is done automatically. In GoAhead, the placement can be done automatically, the user may in addition also manually place the connection macros.

Module relocation in PlanAhead is not supported. For each partial area the user wants to run a module in, a bitstream is required for each module. Integrating n partial areas into a design that each can host all of the m partial modules results in $m \times n$ different bitstream. Following our flow however, only one bitstream per partial module is required as partial modules can be freely relocated within the partial area as well as among different partial areas. Relocation in our systems is carried out by a driver that will be released together with GoAhead.

In contrast to PlanAhead, our flow allows to instantiate a module multiple times within the same partial area (e.g. for connecting modules in a streamlined manner) or within several partial areas. Following the PlanAhead flow, again the multiple instantiation of modules at different positions requires one bitstream for each permutation of a module and its position.

As outlined in Section II, it is beneficial to completely encapsulate the static design and the partial modules. This encapsulation is achieved in the GoAhead flow. The PlanAhead flow however, requires the user to reroute all partial modules after any change in the static system. In addition our flow allows a user, to implement a partial module and later to integrate this module into a already running system. The complete encapsulation of the static system and the partial modules in our flow however favors synthesis and place & route time for modules.

The present GoAhead flow can also be used to implement an island style system. For an island style system each configuration is considered as a module. If two configurations contain the same module, the resulting partial bitstreams however will contain redundancies as both bitstream share the implementation of the common module. Both PlanAhead and GoAhead support building island style systems (as demonstrated in the case study, see also Section V). While PlanAhead only supports island style, in GoAhead a partial area may also host several modules at the same time (slot style system). Within one partial area, we allow to freely place a module with respect to then underlying resource layout (1D slot style). By cascading several partial areas on top of each other, a 2D slot style can be achieved [8].

Neither our placeholder scheme nor the presented migration flow is bound to a particular device. Hence, our tool will also support future devices.

VII. Conclusion

In this paper we outlined how to migrate an existing static only design into a system with partial runtime reconfiguration. The therefore required steps with our tool GoAhead were presented in detail. With our tool, partial runtime reconfiguration becomes available for the low cost Spartan-6 family. GoAhead allows to build sophisticated systems with partial runtime reconfiguration. The tool will be released in the near future and will then also support Xilinx Virtex-5 FPGAs.

Acknowledgment

This work is supported in part by the Norwegian Research Council under grant 191156V30

References

[1] D. Koch, C. Beckhoff, and J. Teich, "A communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 fpgas," in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 253–256.

[2] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, Heidelberg, Germany, Sep. 2008, pp. 119–124.

[3] M. Ullmann, M. Hbner, B. Grimm, and J. Becker, "On-demand fpga runtime system for dynamical reconfiguration with adaptive priorities," in *Field Programmable Logic and Application*, 2004, vol. 3203, pp. 454–463.

[4] Christopher Claus, Bin Zhang, Michael Hübner, Christoph Schmutzler, Jürgen Becker and Walter Stechle, "An XDL-Based Busmacro Generator For Customizable Communication Interfaces For Dynamically And Partially Reconfigurable Systems," *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, Mai 2007.

[5] D. Koch, C. Beckhoff, and J. Torresen, "Demo Paper: Advanced Partial Run-time Reconfiguration on Spartan-6 FPGAs," in *Proceedings of International Conference on Field-Programmable Technology (ICFPTÍ0)*. Beijing, China: IEEE, 2010, to apear.

[6] C. Claus, R. Ahmed, F. Altenried, and W. Stechele, "Towards rapid dynamic partial reconfiguration in video-based driver assistance systems," in *Reconfigurable Computing: Architectures, Tools and Applications*, 2010, vol. 5992, pp. 55–67.

[7] Xilinx Inc., *The Xilinx Design Language*, July 2000.

[8] D. Koch, C. Beckhoff, and J. Torresen, "Zero Logic Overhead Integration of Partially Reconfigurable Modules," in *SBCCI'10*, Sao Paulo, Brazil, Sep. 2010.

[9] Xilinx Inc., *Early Access Reconfiguration User Guide*, March 2006.