Bitfile Preservation - Generation Of Reusable Out Of Context Modules

Christian Stüllein, Norbert Abel, Udo Kebschull
Infrastructure and Computer Systems for Data Processing (IRI)
Goethe University - Frankfurt/Main, Germany
Email: stuellein@iri.uni-frankfurt.de, abel@iri.uni-frankfurt.de, kebschull@rz.uni-frankfurt.de

***Abstract*—This paper presents the idea of *bitfile preservation* which enables the re-use of partial bitfiles in different environments and at different positions of the FPGA without any re-compilation. This way, the behaviour of a system can be changed on the fly just by plugging together different pre-compiled modules (represented by partial bitfiles). These modules can be developed without any knowledge of the final surrounding system. They may even be provided by third-party vendors as closed-source components. Current implementations demonstrate that it is possible to enhance the standard partial reconfiguration vendor flow in a way that enables *bitfile preservation*. Moreover, the already established cooperations prove that *bitfile preservation* is highly relevant for contemporary FPGA industry.**

I. Introduction

As the increase of the frequencies of CPUs came to an end in the beginning of the century, parallel computation became more important to achieve further performance boosts. This technology became known not only in the world of engineering and research but also by end users who are interested in technical details. Unlike the fixed hardware architecture of ASICs such as multi-core CPUs (up to 16x) and many-core GPU's (up to 2048x), where the functionality is defined by software, the flexible reconfigurable architecture of FPGAs offers the possibility for hardware acceleration of certain applications which is based on their inherent parallelism (examples are video processing or data acquisition in high energy physics experiments [1]).
In order to create a design for a specific application, the FPGA engineer following the standard vendor workflow usually has to write or to adopt a lot of code (VHDL or Verilog) and has to go through a complex implementation process. If changes (even small ones) are to be carried out, the complete workflow has to be iterated. This can be troublesome and very time-consuming, especially in the cases where achieving timing closure in some critical components is difficult. This process can be significantly improved by using a module-based approach like the Xilinx design preservation approach [2]. One step further, if pre-tested modules in the form of partial bitfiles are available, an application can be defined (or adjusted) by simply combining these modules. Assuming that appropriate static infrastructure is already programmed into the FPGA, the resulting system can immediately be tested or used by loading these modules into corresponding partitions. Such a solution depends on the presence of freely loadable modules and an appropriate static infrastructure which allows their flexible placement. When following this approach, it even becomes possible to engage third-party vendors to deliver ready-to-use modules. Also, if there already exists many work on this topic (section II-A) they are just few real world applications using this approaches. This partly results from the incompatibility with current FPGAs and vendor tools. Another reason could be that they are difficult to implement or hard to integrate in an industrial application. Since our approach mainly uses the standard vendor toolflow it is easy to integrate and enables the module based approach to a wider range of engineers beyond the scientific community.

Section II shows some of the works dealing with partial reconfiguration and addresses the hierarchical design methodology of Xilinx and Altera, which is the main prerequisite for handling the module netlists separately from the static part of a system. Also the current standard partial reconfiguration flow is briefly described, as it represents the base technology, required to create multiple modules for a dedicated area within the FPGA. In section III we introduce the idea of *bitfile preservation*, which allows the use of modules in different static designs (and different locations) by using the partial bitfile of a module without the need for additional implementation steps. While section IV shows implementation specific details which need to be considered, section V outlines an actual industry application. The conclusion in section VI finally gives a short summary.

II. State of the art

A. Previous works

There is a variety of papers addressing partial reconfiguration, module based approaches and core relocation [3], [4], [5], [6], [7]. Especially the ReCoBus [8] introduced new innovative approaches to design a flexible slot based system supporting the outdated Spartan-3, Virtex-II and Virtex-II Pro devices. It comes with a graphical user interface to support the designer in floorplanning and the implementation of the communication structure (shared bus and streaming bars). The recently released framework GoAhead [9] also

This work was supported by the Federal Ministry of Economics and Technology (BMWi: KF2828501LF1) and the Federal Ministry of Education and Research (BMBF: 16M3205C), Germany.

comes with support for recent Xilinx FPGAs. The latter tools solve the problem of avoidance of feed-through routes (described in section IV) by the introduction of blocker macros (consuming all available routing resources within a defined region). As this macros are created using the XDL (ASCII based netlist representation) representation of a mapped netlist, one can not use the standard Xilinx partition-based DPR workflow, because the partition information will be lost during the XDL conversion. Therefore these tools introduce a completely different workflow the designer is faced with. The deviation from the standard flow brings some disadvantages. As the timing-driven *par* router will ignore the blocker net, the FPGA Editor router has to be used, which may lead to longer compilation times and worse timing results [7]. Furthermore,. the blocker insertion after the map phase can lead to unsatisfactory routing results, because the placer has no information about the subsequently introduced routing restrictions. The workflow presented in this paper is mainly oriented on the standard Xilinx DPR Flow (described in the next subsections), adding some important constraints to achieve the context independence of the modules from the static system.

B. Hierarchical Design Methodology

With introduction of the hierarchical design approach, Xilinx [2] and Altera [10] address some of the problems which are described above. The Xilinx as well as the Altera synthesis flow allow the application to be split up into several partitions. The functionality of these partitions can be described and compiled autonomously. This strategy allows various engineers to work on their parts of the design independently (this approach is termed *Team Design* [10], [12]). In a later step, the resulting netlists are merged to assemble the whole system. Another very important advantage of this method is the design preservation capability [10], [11]. Hardware designers are familiar with the problem of achieving timing closure in timing critical systems — and how disadvantageously it is, when a small change in a non-critical part (e.g. slow control) leads to the loss of timing closure in an entirely different part of the design. The hierarchical design approach provides strategies to avoid this situation by the preservation of satisfactory implementation results. If no changes were conducted in a partition, the already achieved results of a former compilation process can be imported. Through this, the routing (and therefore the timing) of the imported partitions remains untouched. Design preservation can even eliminate the necessity to (re-)verify the untouched parts of the design. As we mainly work with Xilinx FPGAs, all the following descriptions are based on the tools of that vendor.

C. Partitions & Preservation Levels

A typical compilation process is shown in Fig. 1. The design preservation is based on partitions. The designer can define a partition and assign a netlist describing its functionality. One also can set the implementation state for each of the defined partitions either to *implement* or *import*. When set to *import* the tools use the results (synthesis, mapping or routing) of an earlier implementation for that partition. In that way the implementation results of parts of the whole system can be preserved. A possible drawback is the fact that optimizations cannot cross partition borders. However, if this is a problem in an particular application it will always be a task of the designer to decide if the advantage of modularity outweighs this potential drawback.

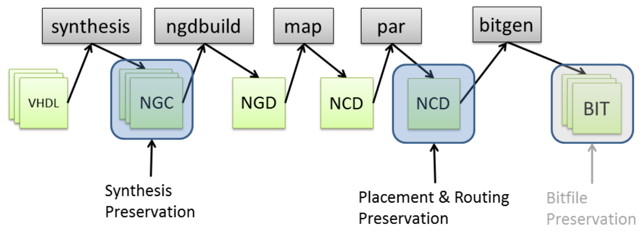

Figure 1. Standard workflow and design preservation: The different implementation tools subsequently add information (technology mapping, placement, routing) to the netlist files which are encoded to a downloadable bitstream file in the final step.

D. Partial Reconfiguration

The term *Partial Reconfiguration* (PR) indicates that only single parts of the FPGA are (re-)configured while the rest of the system remains unmodified. The reconfiguration is performed by loading a partial bitfile (containing the new configuration of the parts to be altered) as if it were a normal full bitfile. Both Xilinx and Altera utilize partitions and routing preservation to generate the partial bitfiles. The key difference between a normal flow and a PR flow is that while using a PR flow a placement area has to be assigned to each partition that shall represent a partial module, and that one can assign more than one netlist to this area (this way, one can decide at run-time which functionality shall be loaded to the area — see Fig. 2).

E. What is still missing

Although both Xilinx and Altera utilize partitions and design preservation to describe partial reconfiguration, the final step (*bitfile preservation*) is still missing. Thus, the last part of the compilation process (the generation of the bitfile) has to be performed in any case, even if the according partition has not been changed at all. Furthermore, if a specific functionality (e.g. Module A) is assigned to a number of placement areas, it has to be translated for each area separately. Using M modules on A areas leads to M implementation runs per area resulting in the potentially huge number of M·A (partial) bitfile generation processes. Additionally, the created partial bitfiles only work in the context (static system) in which they were created. This situation

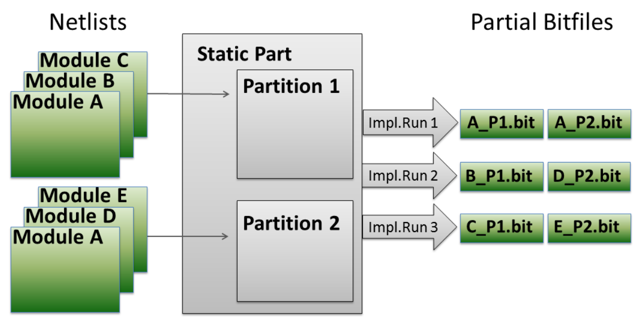

Figure 2. Standard Partial Reconfiguration Flow: by assigning multiple netlists to an area, partial bitfiles can be created during multiple implementation runs. The static part including the module interfaces is routed in the first implementation and imported in the subsequent runs. In this example module A is assigned to both partitions, so that the functionality of this module can be loaded to both locations.

could be significantly improved by using *bitfile preservation* — which is explained in the following section. Xilinx is also facing this issue and introduced the new OOC (Out Of Context) flow [14] in the recently released Vivado Design Suite. The OOC flow is still beta and supports only the latest FPGA family, but it shows that Xilinx has recognized the gap in their current design preservation approach. Compared to the OOC flow, the *bitfile preservation* approach brings additional support for limited module relocation and is not restricted to the series 7 FPGAs. Due to its proximity to the standard DPR flow it is possible to easily adapt it to the Vivado OOC flow to benefit from the new tools.

III. Bitfile Preservation

The idea behind *bitfile preservation* is to generate partial bitfiles which have to be re-created only if their functionality has changed. A change of the surrounding system (if not affecting the module interface), a change of other partitions, or a relocation of the module itself would not lead to a new compile process. As a consequence, the partial bitfile has to be generated only once (instead of A times for A areas), reducing the number of required implementation runs to approximately M/A. Such an approach leads to the situation that already compiled bitfiles can be connected together without any further compile processes.

Hence, the development of the surrounding system and the development of the modules can become processes, which are completely independent from each other. This allows for a module to be designed by a third-party vendor who has no deeper knowledge of the static part of the system and other modules.

Of course, such independence comes with some additional constraints regarding the creation of the partial bitfiles. First, one has to define a number of areas where the partial bitfiles can be placed. These areas have to be identical regarding the footprint (the underlying fabric elements and their arrangement) as well as the interface between the modules and the surrounding system. Second, an area is only allowed to contain routes that belong to the according module. Feed-through routes are not allowed in any case. Third, one has to keep in mind, that the timing characteristics of the FPGAs basic elements can slightly differ from area to area. The next section of this paper focuses on these problems and describes possible solutions.

IV. Implementation

The ML-605 EVAL board (Xilinx) is being used for all our current implementations. The board is equipped with the Virtex-6 LX240T, able to perform partial as well as dynamic reconfiguration in combination with the Xilinx command line tools (version 13.4-14.4) and partial reconfiguration license. For bitfile relocation the RapidSmith framework (version 0.5.0) was used (described in section IV-D).

A. Basic Prerequisites

A few prerequisites need to be met in order to enable the use of partial bitfiles in different environments as well as their relocation. All areas, where the modules may be placed, must be equal in size and shape (there are alternative slot based approaches like [8], [9]). Furthermore, the underlying fabric elements and their arrangement need to be identical. To ensure that the modules can be programmed into the FPGA completely independent from each other, they are not allowed to share a single reconfiguration frame (which comprises one clock region in height).

The interface between the static part and the modules must be defined in a very strict way. In order to enable the reuse of the modules, it is fundamental that the interface is physically identical for all partitions.

Feed-through routes are not allowed at all. These are routes belonging to the static part of the design but crossing a partial area. The presence of such routes causes the system to become inconsistent, when loading a module without the appropriate feed-through route to a location where this route is required.

If these prerequisites are met, it becomes possible to reuse the pre-compiled modules on different locations even in completely different environments (the static part), as long as the module interface is unaffected.

Beside these obvious requirements, some obscure ones have to be kept in mind. Although the underlying fabric elements of the modules are equal in type, small differences in their timing behavior may be present. One example is the temperature drift which varies between the locations. Another example is the correlation between the carry chain and the clock net. The carry chain in every case heading from bottom to top of the FPGA while the clock is distributed upward on the top and downward on the bottom half of the chip — which finally leads to a difference in timing behavior between the bottom and top of the FPGA.

In the current Xilinx partial reconfiguration workflow, the designer can assign netlists of several modules to a partition. Proxy logic (a simple LUT) is automatically inserted by *ngdbuild* for every connection (between the static part and a module) and placed within the partition. The modules are implemented in subsequent runs, while the static part is implemented only during the first one and imported in the other runs (using routing preservation). Because the proxy logic belongs logically to the static part, it is also imported and, therefore, fixed for all modules of that partition. However, since the placement and routing of the proxy logic differs for each partition, Xilinx's standard partial reconfiguration workflow leads only to an intra-module exchangeability (one can load different modules which have been implemented on that specific partition). It does not provide any inter-module exchangeability (one cannot load a module implemented for one area into a different area). If modules are to be exchanged between the areas, the interface must be unified for *all* partitions. This also applies if the modules should be re-used if the static system is changed.

B. Out of context module implementation

As described above, it is required that no elements of the static system are placed within the module regions and no static routes cross that region. The first requirement is satisfied automatically by using the mandatory *AREA GROUP* constraints defining the module partitions. The *map* tool avoids placing elements of the static system within that partitions.

The additional application of the undocumented option *PRIVATE=ROUTE* in the *AREA GROUP* definition causes the *par* tool not to use routing resources within the partition for static routes. This option has been tested with a heavily loaded static design and worked well.

C. Module interface unification

To unify the interface of a module, the LUTs of the proxy logic have to be placed in fixed positions (in relation to the partition boundaries) using the *LOC* constraint (slice selection) in compliance with the *BEL* constraint (a LUT is selected within the slice). The names of the proxy logic instances can be exported using the Xilinx command line tool *pr2ucf*. Additionally, the *LOCK_PINS* constraint has to be applied in order to avoid variations in the used LUT inputs (caused by the optimization process).

Furthermore, one has to register each connection within the static part but close to the border of the partition. These registers also have to be *LOC* and *BEL* constrained, in order to place them right next to the proxy slices. These registers are required to keep the timing results valid over all possible module positions and are also used to define the static system end of the interface routing. Since the slices in the Virtex-6 device contain four LUTs and eight flip-flops, we decided to combine one register slice and two proxy logic slices to build an *interface block* (replaces synchronous busmacros in the old EAPR-Flow) which would allow eight signal lines to pass. Finally, the routing between the PROXY- and Register slices of the interface blocks needs to be unified. This can be achieved by the use of DIRT (Directed Routing) constraints. Because these routes belong to the static part, the corresponding constraints have to be applied only if the static part is implemented. A routing example can be seen in Fig. 3.

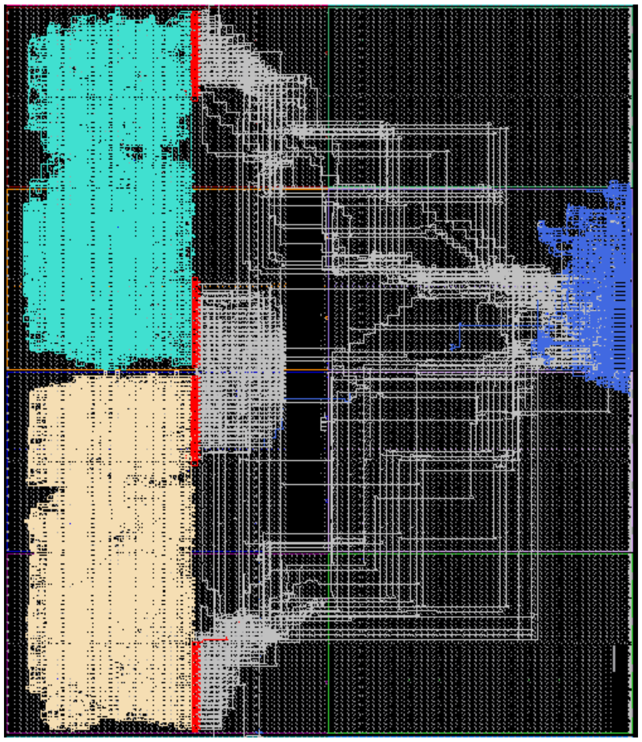

Figure 3. Implementation of the static system and two modules. The modules (turquoise and wheat) are connected through unified interfaces (red) to the static routes (silver), leading to the PCIe block (blue).

D. Bitfile relocation

After the implementation using the standard vendor tools enhanced by the *bitfile preservation* methodologies illustrated before, we acquire one partial bitfile per module, each containing the fabric configuration for a specific location within the FPGA. Thus, the relocation of such a module depends on the change of the address information inside the partial bitfile. Fortunately, this information belongs to the well-documented [13] part of the bitfile and can thus indeed be changed without any recompilation. The change can be done on the fly on PC side (if the configuration process is managed by the PC) or within the FPGA using a special configuration logic as shown in [3]. We are currently

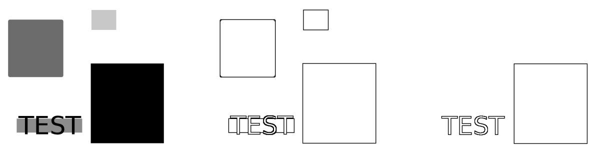

Figure 4. Results:
a) source image without noise b) sobel c) threshold - sobel.

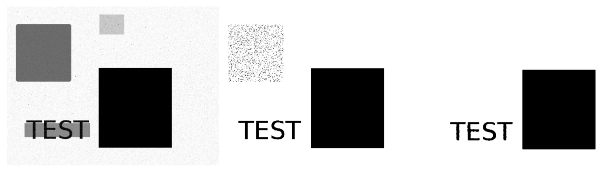

Figure 5. Results:
a) source image with noise b) threshold c) gaussian - threshold.

using a Java tool based on the open source framework RapidSmith [15] to generate the relocated partial bitfiles as an additional step prior to programming them to the FPGA. RapidSmith offers a collection of classes to handle XDL (Xilinx Design Language) files [16], which is a textual representation of the *ncd* files (see Fig. 1). Additionally, this framework provides some classes to work with bitfiles. Our tool utilizes these classes to adjust the address information of the packages within the partial bitfiles. To this end, the row and column addresses of the FAR (Frame Address Register) have to be changed to point to the desired location. Also the CRC for the bitfile is recalculated and updated. The actual configuration data of the fabric elements remains unchanged. As relocation implicitly requires areas with equal size, this potentially leads to waste of unused logic if smaller modules are loaded to the partition, but in many cases (e.g. the application outlined in section V) the benefit of directly loadable modules overweights this drawback.

E. Testdesign

In order to examine the workflow, a test design performing some image processing algorithms comprising two partitions connected together via a pipeline structure was implemented. The static part includes a PCIe block for PIO (Programmed Input/Output) communication, the interface registers for the two partitions and an output FIFO to buffer the processed pixels. The original image is sent line by line over the PCIe link (125MHz) to the FPGA, fed to the pipeline, processed by the modules (250MHz) and read back to the PC. The partition interface consists of a 128 bit wide pixel data bus (16 times 8bit pixels in parallel) and a number of control lines.

For test purposes three different modules were created. These are a gaussian smoothing filter (M1), a sobel edge detector (M2) and a simple threshold binarisation (M3). Since we possess two partitions and three modules (plus one additional *empty* module which only transduces the signals), only two implementation runs are necessary to create all required partial bitfiles. The partial bitfiles for the respectively other area were created by using the bitfile relocation Java tool (described in section IV-D)

In order to demonstrate the *bitfile preservation*, we applied different combinations of the modules using two different static systems. During the test we performed dynamic partial reconfiguration (using iMPACT) to change the execution order of the modules by loading their (relocated) partial bitfiles to the FPGA.

F. Results

Starting with the source image (see Fig. 4 a) we loaded the sobel module M2 to the first position, leaving the second one empty. The result is displayed in Fig. 4 b. In order to get the text ”TEST” more clearly, we subsequently changed the processing by loading the thresholding module M3 to position 1 and moving the sobel module M2 to the second position. The result can be seen in Fig. 4 c.

A second test used the original source image with added noise (see Fig. 5 a). If the threshold module M3 was applied directly, we mistakenly received some black pixels from the upper left block as can be observed in Fig. 5 b. After modifying the configuration by loading the gaussian module M1 to position 1 and the threshold module M3 to the second one, we obtained the expected result (see Fig. 5 c).

The tests were additionally repeated with a modified and re-implemented static system (but re-using the placement and routing constraints for the module interfaces). Also when using the originally created (partial) bitfiles of the modules with the new surrounding system - the design still works and leads to the same results.

V. Applications

This section focuses on the industrial usage of *bitfile preservation* and how it can improve real world applications. For this, lets take a closer look at the Silicon-Software company, which is located in a typical field for FPGA applications — the processing of video streams.

Designers usually have to describe the complete system including the different processing units in a description language on the Register Transfer Level (RTL). Since this is very complex, time-consuming, error-prone and therefore expensive, Silicon-Software focuses on the creation of a more comfortable solution to create such streaming applications. In addition to specialized hardware (microEnable) they provide a software solution to create FPGA designs. While focusing on image and video processing, the graphical user interface VisualApplets (see Fig. 6) lifts the design to a higher system level, where the designer just combines predefined processing elements to describe the application.

Although these products simplify the design of such applications, there is still the requirement for a complete

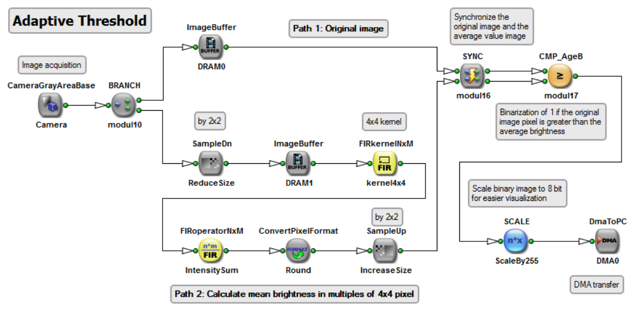

Figure 6. VisualApplets GUI for system and module definition [Silicon-Software].

implementation run before the system can be used, even if only small changes are made. This issue is currently addressed in cooperation with Silicon-Software. With the presented *bitfile preservation* approach it becomes possible to create or deliver ready-to-use partial module bitfiles. The designer can create the application specific algorithm (the static system) and additionally define some reconfigurable modules (e.g. for different preprocessing stages) within the processing pipeline. After the static design is implemented and loaded to the FPGA different pre-compiled modules can be loaded to the defined positions at runtime to adjust the algorithm. Using *bitfile preservation* this modules can be used without the need for any recompilation to adjust them to the current static system. The end user will benefit from the minimized delay between graphically (re-)defining the application and running the reconfigured system. In this way, it becomes possible to test various combinations of filters until the best setup is found without any additional recompilation steps.

Beyond that, the user can create his own modules using the graphical editor and add them to the library for later re-use. A major improvement is the possibility to include encrypted bitfiles from third party vendors in the library. When using *bitfile preservation*, these modules are independent from the surrounding system and do not need to be recompiled for the actual application specific static system. In summary, *bitfile preservation* considerably improves the flexibility of FPGA designs and thus significantly speeds up the search for an optimal filter setup in video processing.

VI. Conclusion

This paper presented the concept of *bitfile preservation*. The standard implementation flows of both Xilinx and Altera already contain preservation methodologies (such as routing preservation) which have proven to be helpful in many real world applications. Even though both companies focus on partial reconfiguration, the final preservation step *bitfile preservation* is still missing. In this paper we have shown that there exist several problems which have to be solved before *bitfile preservation* can really be used. However, we believe that these constitute only some minor changes of the partial reconfiguration flow, when compared to the considerable benefit coming from *bitfile preservation*. Accordingly, current cooperations between our research group and industry have already proven the relevance and the significant improvements stemming from this approach.

References

[1] Abel, Gao, Kugel, Meier, Männer, Kebschull (2009) DATE09 conference, *DPR in CBM: an Application for High Energy Physics*

[2] Xilinx (2010) User Guide, *UG748: Hierarchical Design Methodology Guide (v 12.3)*

[3] Ferrandi, Morandi, Novati, Santambrogio, Sciuto (2006), *Dynamic Reconfiguration: Core Relocation via Partial Bitstreams Filtering with Minimal Overhead*

[4] Sedcole, Blodget, Becker, Anderson, Lysaght (2006) IEE Proc.-Comput. Digit. Tech. Vol. 153 No 3, *Modular dynamic reconfiguration in Virtex FPGAs*

[5] Majer, Teich (2007) VLSI Signal Processing, Vol. 47, *The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer.*

[6] Edson L. Horta, John W. Lockwood, David E. Taylor, David Parlour (2002) Design Automation Conference (DAC), *Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration.*

[7] Ali Asgar Sohanghpurwala (2010), *OpenPR: An Open-Source Partial Reconfiguration Tool-Kit for Xilinx FPGAs.*

[8] Koch, Beckhoff, Teich (2008) FPL2008, *ReCoBus-Builder a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs.*

[9] Beckhoff, Koch, Torresen (2012) FCCM2012, *GOAHEAD: A Partial Reconfiguration Framework.*

[10] Altera (2011) Handbook, *Quartus II Incremental Compilation for Hierarchical and Team-Based Design (v 11.1)*

[11] Xilinx (2010) White Paper, *WP362: Repeatable Results with Design Preservation (v 1.0)*

[12] Xilinx (2011) White Paper, *WP388: Increased Productivity Using Team Design (v 1.0)*

[13] Xilinx (2011) User Guide, *UG360: Virtex-6 FPGA Configuration (v 3.4)*

[14] Xilinx (Oct. 2012) User Guide, *UG905: Hierarchical Design: Design Reuse (2012.3)*

[15] Lavin, Padilla, Lamprecht, Lundrigan, Nelson, Hutchings (2011) FPL2011 conference, *RapidSmith: Di-It-Yourself CAD Tools for Xilinx FPGAs*

[16] C. Beckhoff, D. Koch, J. Torresen, *The Xilinx Design Language (XDL): Tutorial and Use Cases*

[17] P. Sedcole, B. Blodget, T. Becker, J. Anderson and P. Lysaght, *Modular dynamic reconfiguration in Virtex FPGAs*