EasyPR – an Easy Usable Open-Source PR System

Mr. and Ms. Blind
some hidden place
Email: {Mr_Blind, Ms_Blind}@blinded.org

***Abstract*—In this paper, we present an open source partial reconfiguration (PR) system which is designed for portability and usability serving as a reference for engineers and students interested in using the advanced reconfiguration capabilities available in Xilinx FPGAs. This includes design aspects such as floorplanning and interfacing PR modules as well as fast reconfiguration and online management. The system features relocatable modules which can even contain reconfigurable modules themselves, hence, implementing hierarchical PR. In addition to free access to all design files, the system can be tried out remotely via the Internet without the need to install tools or having access to a physical board.**

I. Introduction

Partial runtime reconfiguration (PR) of FPGAs has been demonstrated in several applications, mainly for saving area. By using smaller FPGAs, cost and power can be significantly reduced (the smaller an FPGA, the lower its static power). For instance, in [1], [2], [3] PR is used for adapting the systems to different environments (e.g., light conditions for a vision system or channel quality for a wireless communication system). In [4], partial reconfiguration is used in a database accelerator to compose SQL operator modules dynamically together to execute SQL queries. This system uses module relocation and multi-instantiation of modules which is not available in the Xilinx vendor tools. Common for all these examples is that only the entire active modules are loaded to the FPGA instead of having all modules fitted together on a (larger) device.

While the benefits of PR are obvious, it is still difficult to make use out of this technology. The FPGA vendor tools support only very simple application scenarios (see also Section II-A) and academic results are very often not made freely available. However, with OpenPR [5] and BLINDTOOL [6] there exist available tools allowing to exploit more advanced PR features than what is provided by the FPGA vendors. With EasyPR, we want to push this further by providing an out of the box PR-system including all components as open-source.

The overall goal of the EasyPR project is to provide an example system serving as a template for building advanced run-time reconfigurable systems. Consequently, we aim at portability among different FPGA families and generic design practices allowing for fast reuse. So far, we only support devices from the FPGA vendor Xilinx. However, the here presented concepts are very general and might be used with other FPGAs with the availability of corresponding tools.

In many cases, only bounding boxes of reconfigurable regions and the number of input and output wires have to be adjusted in the provided scripts in order to generate all additional constraints that are necessary for implementing a reconfigurable system. Furthermore, we provide a portable lightweight configuration manager which can be easily controlled by a CPU or by a dedicated control state machine.

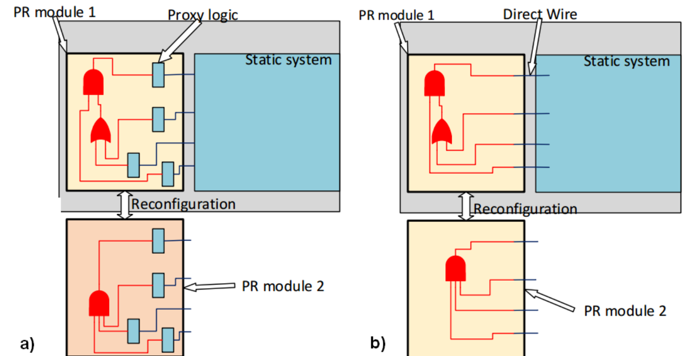

Fig. 1. Single island style reconfiguration. a) when using the Xilinx proxy logic approach, route-through LUTs are placed inside the island and connected to the static system in an initial implementation step (blue wires). Based on this, each module is implemented in an incremental design step (red wires). b) direct wire linking does not need proxy logic. The direct interface wires are strictly constrained and not determined by the router as in a).

II. Reconfiguration Styles

It is important to understand that there is not one specific way to build a reconfigurable system and depending on the application, various approaches exist for building a PR system. For example, for bootstrapping, only one single PR module will be used that is loaded after an initial time-critical configuration step. In other scenarios, several different or identical modules might be placed together in one combined region. A classification of such different scenarios was presented in [7] under the term *configuration style*. In that publication, three styles were mentioned. The basic configuration style where only one module can be placed exclusively in a reconfigurable region is called *island style*. In the *slot reconfiguration style*, multiple modules can be placed in a one-dimensional manner. The most flexible configuration style, where several modules can be placed freely in a two-dimensional grid, is called *grid style*. The different configuration styles have individual requirements, in particular in the way the communication has to be implemented with the reconfigurable modules. Note that different configuration styles might be mixed or used several times in a system. EasyPR provides examples for all three configuration styles, as described in the following sections.

A. Single Islands

Placing reconfigurable modules exclusively in a single reconfigurable region is the only configuration style supported by the FPGA vendors Xilinx [8] and Altera [9]. The Xilinx

PR flow is illustrated in Figure 1a). As the routing between the static system and a reconfigurable module is not constrained, it is not possible to reuse a partial module in a different island (neither inside another island within the same system nor in any other system). This holds even if all island shapes are identical (in terms of CLBs, BRAMS, etc.). Moreover, any change in the static system may reroute the paths to the proxy logic, hence, demanding to also reroute all partial modules. As a consequence, this flow is rather cumbersome when using several islands and modules. For example, a system with 4 islands and 5 different partial modules (placeable in all islands) needs 20 partial routing steps when modifying the static system.

In EasyPR, we constrain the routing between the static system and the partial modules to fixed wires as shown in Figure 1b). While this needs an extra step to constrain the signal-to-wire binding, it completely removes the dependency between the static system routing and the partial modules. In EasyPR, partial modules can even be implemented before the static system. The signal-to-wire binding is implemented using the BLINDTOOL flow [6] where special blockers are generated to occupy all routing resources that are prohibited in a specific routing step (e.g., all wires (except the direct connection wires) inside the island, when implementing the static system).

EasyPR comes with generic BLINDTOOL scripts where it is sufficient to define 1) reconfigurable regions, 2) the areas where to allocate the direct interface wires and 3) the signal names for the interface wires (which are typically the top-level names of the partial module). This process is supported by a user-friendly GUI. With this information, BLINDTOOL creates all necessary UCF constraints, a wrapper for the static system, a wrapper for the partial system and special blocker modules for constraining the routing. The BLINDTOOL script debugger allows observing all steps performed by the script. The physical implementation of the static system or the partial modules is carried out by a batch script calling the Xilinx tools and performing all necessary transformations (e.g., inserting the blocker into the design).

The wrappers, which are generated by BLINDTOOL act as placeholders. When implementing the static system, we use a wrapper acting as a placeholder for the partial module. Similarly, partial modules connect to a wrapper which replaces the static system.

While BLINDTOOL allows a very flexible resource allocation, we restricted design parameters in EasyPR to favor usability and portability (keep it simple!). This includes:

1) the module height has to be defined in full clock regions
2) a module must contain at least one CLB column
3) reconfigurable regions shall not contain I/O pins or clock primitives (e.g., BUFGs, DCMs, etc.)
4) interface wires are predefined (EasyPR uses double wires for connecting PR regions that are available on all Xilinx FPGAs) and up to 4 input signals and 4 output signals can be connected per CLB wherof the direction (north, east, south, west) has to be identical for all wires.
5) the partial subsystem uses only one clock domain
6) no static routing allowed inside a reconfigurable region

In most cases, these rules are easy to follow and many of them are mandatory in other PR flows. For example, using the Xilinx PR flow demands the rules 1), 2), and 3) and OpenPR [5] needs pretty much the same rules, except that the old bus macro communication is used for the interface signals rather than the more efficient and flexible direct wire binding, which is uded in EasyPR(rule 4). However, BLINDTOOL is not bound to any of these constraints and experienced users may override some or all rules (e.g., by designing a reconfigurable module which only contains BRAM content or for routing static signals through a reconfigurable area). Again, EasyPR was made to provide a smooth quick start into run-time reconfigurable system design by removing as many obstacles as possible, while still providing many powerful PR features.

B. *Slot Style and Grid Style Reconfiguration*

EasyPR provides examples where multiple modules can be placed in a combined reconfigurable region. The flow is very similar to the single island style, except for some additional rules:

1) the module/region width has to be an even number (2, 4, 6, ...) in terms of CLB, BRAM, or DSP columns
2) module placement is vertically restricted in terms of clock regions and horizontally in steps of two CLB/BRAM/DSP columns.
3) communication is bound to nearest neighbor communication in all four directions for the grid style and to east/west for the slot style.
4) due to the communication scheme from the last rule, interface signals might glitch during reconfiguration.

As with the single island rules from the last section, it is possible to override all these additional rules when using BLINDTOOL. For example, it is possible to reconfigure individual modules glitch free without any interference to other modules inside the same reconfigurable region.

C. *Advanced Examples*

The baseline examples demonstrate the most common PR use cases for single island style reconfiguration as well as for slot style and grid style reconfiguration where multiple modules can be placed in a shared area. In addition, we are providing more advanced examples including custom instruction set extensions where modules can be as small as a single 8 CLB tall subcolumn. We also show how a MIPS CPU that contains reconfigurable instructions can itself be implemented as a reconfigurable module. In this case, the reconfigurable instructions will be reconfigurable modules inside a reconfigurable module, hence, demonstrating hierarchical PR. We also show how to route through reconfigurable regions without loosing the ability of module relocation. While all advanced examples are provided for Spartan-6 FPGAs, we will

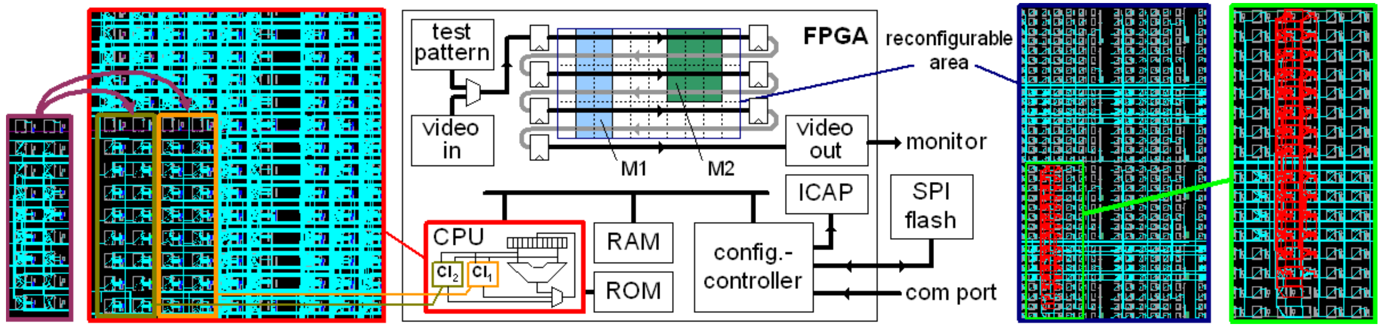

Fig. 2. Xilinx bitstream format. EasyPR reference system with two partial regions for hosting custom instructions in a softcore CPU and a reconfigurable area supporting two-dimensional module placement. The configuration controller supports DMA transfers from an external SPI flash to the ICAP port.

successively port them to other devices.[1]

III. The EasyPR Reference System

EasyPR is built around the BlindTool framework (available under [10]). The tool and the flow is described in [6].

The EasyPR reference systems is currently available on two FPGA platforms: the Spartan-6 Atlys board and the Zynq ZedBoard (not using the ARM core). However, the design can be easily ported to all devices supported by BlindTool (Virtex-6/7, Spartan-6, Zynq, Kintex). The minimum requirements are a VGA output and a com port. For using all features, a video input path and a SPI flash must be available. We kept the reference systems plain for simplifying modifications (e.g., changing reconfigurable regions or adding extra logic). For implementing own partial modules, it is sufficient to instantiate the module in the provided project and to define the module height and the left and right columns of the module on the device. This is supported in the BlindTool GUI. Running the tool flow will produce a partial module that can then be placed into the static reference system. This is again supported by BlindTool that can show all valid placement positions for the new module.

In the following sections, we will introduce the main components provided in EasyPR, except for the configuration controller that we devoted a dedicated section (Section IV).

A. MIPS CPU with Reconfigurable Custom Instructions

The system provides a very simple MIPS-I compatible softcore CPU. The CPU is not pipelined making the code very readable. However, on a Spartan-6, this CPU still runs at 50 MHz which results in 50M instructions peak performance, more than what many microcontrollers provide.

To this CPU, we added two PR regions for hosting custom instructions (CI_1 and CI_2 in Figure 2). This allows more complex instructions to be added to the MIPS and allows the combination of a RISC CPU with the possibility of computing instructions that are more typical for CISC CPUs. As examples, we included the following custom instruction modules:

- count-ones: return number of `'1'`-bits in a 32-bit vector
- CRC: computes a CRC checksum
- bit-permute: [31..0] = [0..31] (uses routing only)
- saturation-add: adding without overflow
- byte-add: 4 8-bit adders instead of 1 32-bit add

The first three instructions all save about 100 original MIPS instructions per call and nicely demonstrate the advantage of customizing the ISA (instruction set architecture). In addition to the custom instructions, we are also providing code examples and a small tutorial on how to announce a custom instructions to the compiler[2] and how to call them in a C program (via inline assembler).

The MIPS CPU takes only 1400 Spartan-6 LUTs including (DSP48) multiplication and barrel shifting, but without division or floating point support. For saving resources, it is possible to remove unused instructions from the CPU, which is called ISA subsetting [11].

B. Reconfigurable Area

The system not only provides a runtime reconfigurable CPU, it also includes a larger reconfigurable area for hosting several relocatable modules. As shown in Figure 2, this area is tiled into multiple identical rows. The columns can include logic (CLB), memory (BRAM), or multipliers (DSP). For the video streaming, a regular structured path is routed in a meander style regularly over all rows. Here, we used a path of double wires (routing two CLBs far) that provide the necessary connections and that are available on all Xilinx FPGAs. This defines tiles to be two CLB/BRAM/DSP columns wide and this is the reason, why modules have two be a multiple of two columns wide. This rule is the key to provide communication with relocatable modules as we can now use equivalent wires on the left and right side for streaming in and streaming out the video data. This is constrained with the help of BlindTool.

In the provided reference system on the Atlys board, the reconfigurable area is 16 columns wide and 5 clock regions

[1] We favor Spartan-6 devices, because they are low cost (boards start below 50 US$) and because the have the finest granularity for reconfiguring the logic among all Xilinx FPGAs. Reconfiguring Xilinx FPGAs can only be done by writing *frames* that contain configuration information for all CLBs in a column within the span of a clock region. In the case of Spartan-6, the smallest reconfigurable update will affect 16 CLBs while it is 40 CLBs for Virtex-6 FPGAs. Another advantage of Spartan-6 FPGAs for PR is that I/Os are located around the die and not in the middle of the fabric (as with recent Virtex-FPGAs). This simplifies floorplanning when using large PR regions.

[2] This is done in the `mips-opc.c` file of the GCC cross compiler. In that file, a new instruction can be added by using an otherwise not used instruction word. After this, the instruction is available in the (dis)assembler.

tall. This area provides in total 1040 CLBs, 40 BRAMs, and 20 DSPs which are available in $16/2 \times 5 = 40$ tiles. While modules can be as small as a single tile (hence, allowing to fit in 40 reconfigurable modules at the same time), it is possible to allocate multiple adjacent tiles depending on the module requirements. The Zynq reference design provides 18 columns in two clock regions resulting in $18/2 \times 2 = 18$ tiles (with 1400 CLBs, 40 BRAMs and 40 DSPs).

C. Partial Module Library

EasyPR comes with reference modules that are provided as source (VHDL) and relocatable netlists (BLINDTOOL netlist format). From the latter, partial bitstreams can be easily generated in BLINDTOOL and tried out on an FPGA without running place and route for the provided modules. The current library contains the following modules:

- BMP-viewer: a small overlay module inserting a bitmap into the video stream
- skin-color: a classifier module marking pixels if they are in the range of human skin color
- Pong: arcade game controlled by the push buttons
- Hex-viewer: tiny module displaying hex values in a box. The number of digits, number of rows, the font size, and transparency can be adjusted by generics.
- Sobel: edge detection as an example for a sliding window operator
- Game-of-life: a partial module consisting of a baseline MIPS CPU, a small overlay frame buffer, RAM and instruction ROM.

All modules can be placed together into one large reconfigurable area. The order in which the modules are placed defines then the layer on the screen (the closer a module is placed to the video output, the further in front it appears on the screen). All modules, except the Game-or-life, come as one source file for simplifying IP reuse.

IV. CONFIGURATION CONTROLLER

EasyPR includes a configuration controller for writing configuration data to the internal configuration access port (ICAP) of the FPGA. During partial reconfiguration, the involved reconfigurable area will not provide any service and in order to minimize this overhead, configuration data has to be written fast. For this reason, the HWICAP module [12], which is provided by Xilinx, is commonly not used for high speed reconfiguration. HWICAP is slow because data is written by a CPU to the ICAP port rather than using DMA [12]. This results not only in weak reconfiguration times, but also comprises extra load for the CPU. Consequently, several configuration controllers have been presented targeting high configuration speed as the main objective.

In [13] internal BRAM memory was used for storing partial bitstreams. This approach is obviously only suited for very few and small modules. [14] proposes a configuration controller using DMA transfers from external SRAM and the configuration controllers in [15], [16], [1] fetch the bitstreams from external DDR memory. The last approaches use the Xilinx DDR memory IP core by adding a state machine issuing DMA requests for transferring configuration data from the external memory to the ICAP port. Because SRAM and DDR memory is volatile, partial bitstreams have to be preloaded (e.g., at system start). Another issue is that many published configuration controllers are not open source. However, [15], [14] are freely available and can also be used with EasyPR.

Opposed to previous approaches, our configuration controller is designed to provide as much reconfiguration data as possible from an SPI flash memory to the ICAP primitive. We chose flash because of the many academic boards which provide a quad-SPI mode flash memory (QIO-SPI). For example, the Spartan-6 Atlys board from Digilent, the Spartan-6 FPGA LX9 Board from Avnet, the Kintex-7 KC705, and the Zynq ZedBoard, include all a QIO-SPI flash memory. In addition to the wide availability, flash does not need to be preloaded at system start and board vendors commonly provide solutions for writing memory images to the flash. Another advantage of configuring from flash is that we do not demand DDR or SRAM memory bandwidth which removes possible interference of a configuration process with the operation of the rest of the system (because of the shared memory bandwidth). We also need no extra access to the DDR memory which in some cases might impact performance (e.g., by adding an extra memory port). Instead of this, our flash memory configuration controller utilizes the (in most cases) unused bandwidth of the flash memory.

The QIO-SPI mode allows to read data as fast as 50 MB/s. We implemented a QIO-SPI DMA controller able to fully utilize this speed. For even further improving speed, we included a decompression module that uses LZSS [17] compression. Bitstream decompression not only allows for faster reconfiguration, it also shrinks the bitstreams, hence allowing to store more reconfigurable modules in the same memory. Note that the LZSS decompression module might also be used for fast system start by decompressing software binaries or other data that is read from the flash.

A. The Flash Configuration Controller Hardware

The block view of the configuration controller is shown in Figure 3 and consists of the following components:

- Flash controller.
- SPI flash reader.
- Decompression module.
- UART receiver.
- 8-bit to 16-bit (or 32-bit) buffer.
- ICAP primitive.

1) Flash controller: This module controls reading configuration data from the flash memory. It has 3 registers accessible on the system bus:

- *Address register* (24-bit): The read address which is sent to the flash memory (bitstream memory location).
- *Size register* (24-bit): Number bytes to read from flash memory (bitstream size).
- *Control register* (3-bit): Flags for enabling/disabling the configuration and for checking the configuration progress.

After writing start address and size, the controller starts reading data from the flash. This data can be either passed

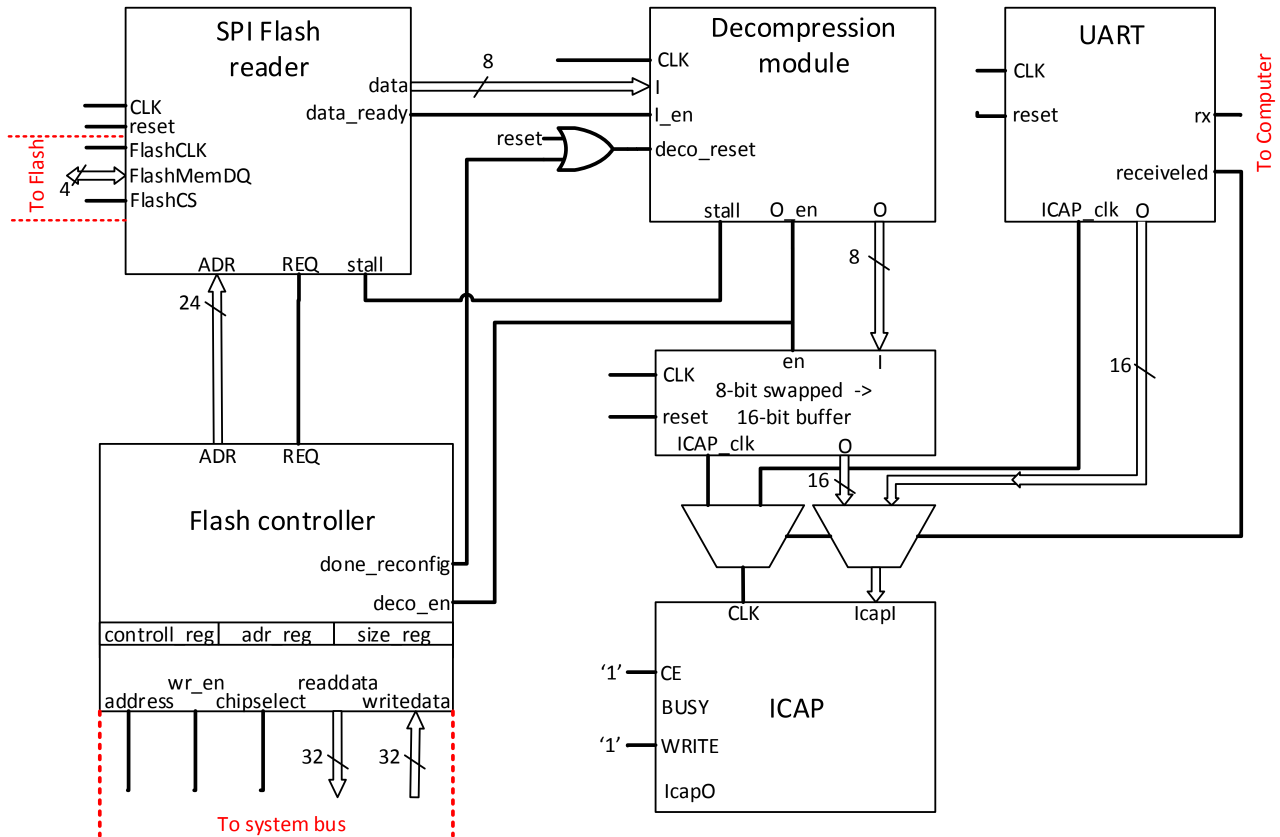

Fig. 3. Block view of configuration controller. The red emphasized signals are the top level signals of the configuration controller while all other signals are internal. The architecture is generic and for supporting different Xilinx FPGAs, word sizes will be adjusted to fit maximum ICAP width.

through the decompression module or sent directly to ICAP. In both cases, the number of data words that are written to ICAP are counted and not the words that are read from the flash memory, which is more intuitive to use. The system bus can be easily connected to any established bus standard or even replaced by some logic writing directly to the registers.

2) SPI flash reader: The SPI flash reader module is designed to read data from a 128 Mbit Numonyx N25Q12 flash memory (or any compatible device) which is available on many popular FPGA boards. The N25Q12 transfers data over SPI at maximum specified clock frequency of 108 MHz. In QIO-SPI mode, the read speed from the memory is 50 MB/s at a memory clock frequency of 100 MHz.

At start-up (or reset) of the system, the flash reader module initializes the high-speed QIO-SPI mode on the N25Q12. After this, a ready flag is set to signal that the module is now ready for QIO-SPI read instructions. In read mode, the flash reader waits for a position-length tuple and a read strobe. On this event, the flash reader sends the corresponding read instruction and memory address to the flash and continues to receive data. The flash memory provides a 4-bit data item each clock cycle until the fast read instruction is terminated by the flash reader. A buffer/register is used to assemble larger words from consecutive nibbles. To temporary halt flash memory reading without ending transfer (e.g., for flow control), an input signal for stalling (STALL) is provided. When a stall request is received, the flash reader holds the flash memory clock signal low to halt transfer. For safely stalling the flash clock, an I/O primitive (ODDR2) (originally intended for double-data transfers) was used to prevent glitches. In addition, the ODDR2 primitive guarantees a low skew on the clock pin of the flash chip. All flash interface signals were constrained with flip-flops in the input/output blocks (IOBs) of the FPGA. This hides timing issues from long signal paths within the FPGA. A user has only to paste the provided UCF constraints into his own design and adjust 6 IOBs locations.

3) Decompression module: At 100 MHz system clock, the flash memory can provide 50 MB/s. Adding compression on the bitstream allows faster reconfiguration, but requires a module for decompression before data is sent to the ICAP. We achieved a compression ratio of about 50% for highly utilized modules and much better values for modules leaving more resources unused. With this, we can basically double the configuration speed from the flash.

The configuration bitstreams are compressed with Lempel-Ziv-Storer-Szymanski (LZSS) [17] lossless compression be-

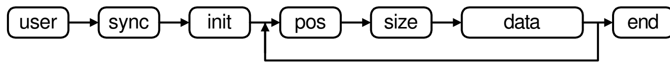

Fig. 4. Xilinx bitstream format. After a preamble, one or more sequences containing the configuration data will follow. Each sequence starts with a position field followed by the actual configuration data. For module relocation, only the position field is adjusted. The figure is taken from [7].

fore they are stored in the flash memory. The decompression module is placed in the configuration controller between the memory and ICAP to decompress the configuration bitstreams. Our decompression module is described in [18] and takes some of the ideas published in [19]. We optimized the decompressor for low area overhead, high decompression speed and good compression ratios.

4) UART receiver: The UART receiver is added to the configuration controller as an option to allow reconfiguration with bitstreams not stored in the flash memory. By default, it operates at 115200 baud. Alternatively, other data rates can be set by a generic value and even Mbit rates are supported for systems providing a corresponding serial interface (e.g., when using a FTDI USB to RS232 bridge). When a bitstream is sent over the UART, this module takes control over the ICAP and starts writing the received configuration data to the ICAP data port. This process will be much slower than configuring from flash memory and was added to support portability for systems without SPI flash memory and also for testing new modules without the need to upload bitstreams to the flash memory.

5) Internal Configuration Access Port (ICAP): To carry out reconfiguration at runtime, the system needs to write configuration data into the configuration cells of the FPGA. For self-reconfiguration on Xilinx devices, this means writing data to the ICAP port which is basically an internal version of the SelectMap configuration port. Our configuration controller uses ICAP at maximum possible word size, which is 16-bit for Spartan-6 FPGAs and 32-bit for all Virtex families.

We are currently not using configuration readback. A future version of the EasyPR configuration controller will contain an optional readback option with state extraction and restoration (for preemptive execution of partial modules).

B. Module Relocation

With EasyPR, we want to demonstrate how to easily implement complex reconfigurable systems on FPGAs. This commonly implies systems where modules can be placed to different positions on the FPGA. For allowing module relocation, we have to physically constrain modules, for example, to fit into bounding boxes. This process is carried out at design time. In addition, we have to modify address information inside the bitstream that defines the physical module position on the FPGA fabric, which has to be accomplished at runtime. A flow chart illustrating the Xilinx bitstream format is shown in Figure 4. The flow is identical for all Xilinx FPGAs but details including word size, encoding of positions, and bitstream sizes might differ among different device families.

For manipulating the placement information, a (signed) displacement value has to be added to the address (this address is called major address (MJA) by Xilinx) which has to be written to the frame address register (FAR). For example, for shifting a module by one CLB column to the right, the MJA address has to be incremented by one. Similarly, there is a row address for relocating modules in vertical direction. The exact format is FPGA family specific and the details can be found in the corresponding user guides. However, the exact bitstream encoding is not officially disclosed by Xilinx and there are some issues that should be understand, when using bitstream relocation:

- *Different blocks* are used to distinguish between different sections inside the bitstream. This could be, for example, the configuration of the CLBs or the initial values of the BRAMs, which are stored in another section. As a consequence, there are multiple displacement values, in the case a module uses multiple sections. For instance, relocating a module consisting of CLBs and BRAMs needs a displacement value for the CLBs and another (different) displacement value for the BRAM content. The displacement values are specific for each FPGA system and depend on the number and position of BRAM columns on the device.
- *Heterogeneous logic resources* might be encoded differently. For example, Xilinx Spartan-6 FPGAs have logic tiles (CLBs) that can either provide arithmetic functions with carry chains (called CLEXL) or other tiles that can provide arithmetic and (in addition) distributed memory (called CLEXM). Distributed memory allows for the efficient implementation of small shift registers or register files. If a module is not using distributed memory, it can be relocated arbitrarily, but the bitstream encoding differs depending on the used CLB type (CLEXL or CLEXM).

As our goal is to provide an easy usable and portable flow, we aim at hiding these low level issues by creating all partial bitstreams for all possible module positions. However, we already know that relocated module bitstreams may be identical except for the FAR address register values (containing the MJA and row addresses). We use this knowledge to significantly save memory for relocatable module bitstreams. If we study again Figure 4, we see that all blocks are fixed except for the POS-field which is containing the position information.

We use this information to derive blocks that are 1) position independent and 2) position dependent as sketched in the following algorithm:

1 : `Input`: {module M_0, placement_positions P_0}, {M_1, P_1}, ..., {M_k, P_k}
2 : `Output`: Bitstream_lists B_0, B_1, ..., B_k,
3 : m =`PlaceModuleToCurrent`(P_0)
4 : b_0 = `GeneratePartialBitstream`(m)
5 : $B_0 = b_0$
6 : $\forall P_i, i > 0$ `do`
{
7 : m = `PlaceModuleToCurrent`(P_i)
8 : b = `GeneratePartialBitstream`(m)
9 : $offset = 0$
10: $length = 0$
11: `while` $(offset + length \leq$ `size`(b)) `do`
{

```
12:    offset = offset + length
13:    length = CorrelateBitstream(b_0, b, offset)
14:      if length > 1       // position independent bitstream
15:        B_i = B_i & AddReference(b_0, offset, length)
16:      else                        // position dependent bitstream
17:        B_i = B_i & AddWord(b, offset)
       }
    }
```

The algorithm is called with a list of all modules with all corresponding possible placement positions. For the first placement position of each module, we generate a complete partial bitstream b_0 and store the result in our bitstream repository (lines 3–5). For all other placement positions, we create a partial bitstream in the same way. We then correlate this new bitstream b with the first bitstream b_0 starting at the position $offset = 0$ (line 12). The result is the length of the longest match where the two bitstreams are identical. If the matching length is larger than one, we append a reference referring to the original bitstream to the configuration for the current placement position; otherwise, we will add the current word from the bitstream b. This process is repeated until the end of the bitstream. Note that we use the native word size of the configuration state machine which is 32 bit for all Virtex FPGAs and 16 bit Spartan FPGAs from Xilinx.

The algorithm is basically a compression algorithm that has some similarities with LZSS [17] but which is specifically tailored to compress mostly identical bitstreams. For relocating the partial module bitstream b_0 to another position, we will 1) take the preamble from b_0 (the blocks `user`, `sync`, and `init` in Figure 4), 2) concatenate one word containing the address information (the block `pos`), and 3) append a final sequence from b_0 (the blocks `size`, `data`, and `end`). In the case a module has very low resource utilization or in the case a module uses logic and BRAM, the sequence will be longer according to the number of position fields in the bitstream.

For clarity, we omitted a discussion about heterogeneous logic resources that have different bitstream encoding (the CLEXL or CLEXM issue mentioned above). Different encodings are handled by storing multiple reference bitstreams and not only a single b_0. This also means that the correlation has to be computed against multiple bitstreams and references can point to different reference bitstreams. In the case of creating bitstreams for Spartan-6 FPGAs, for example, there are situations that demand up to 4 reference bitstreams when relocating modules arbitrary in the left or right half of the device and when allowing module relocation to start at any odd or even CLB column. This is because only every other CLB column provides distributed memory which is differently encoded than the other arithmetic only CLB columns.

1) Implementation: The module relocation is done at design time at the netlist level. This can be easily performed in BLINDTOOL by running the following script:

```
OpenBinFPGA FileName=xc6slx45csg324-3.binFPGA;
AddBinaryLibraryElement FileName=module.mod;
PlaceModule Location=INT_X%X_pos%Y%Y_pos%
            LibraryElementName=module;
SaveAsDesign FileName=module_at_X%X_pos%Y%Y_pos%.xdl;
```

	Spartan-6	Zynq
SPI flash reader	55 (25)	86 (38)
UART receiver	255 (90)	278 (99)
Decompression	63 (39)	67 (49)
Flash controller	145 (44)	202 (78
Configuration controller	518 (189)	633 (264)
Without UART	263 (108)	355 (165)

TABLE I
CONFIGURATION CONTROLLER RESOURCES IN TERMS OF LUTS (SLICES).

We have to set the environment variables `X_pos` and `Y_pos` with the placement position (top-left corner) and BLINDTOOL will then place the module `module.mod` on a completely empty device. The result is a netlist in the XDL (Xilinx Design Language) format. This netlist will be converted into the binary netlist format (.ncd) from which we generate a partial bitfile using the `bitgen -r` option. Here, we will call the Xilinx bitstream generation program bitgen to create a partial bitstream containing the differences to a *completely empty bitstream.*

2) Hardware Support for Module Relocation: In order to allow fast reconfiguration with a minimum of CPU interaction during the reconfiguration process, we added small Fifos instead of a simple register to store the address and size values in the flash controller (Section IV-A1). This extension makes it possible for the configuration controller to chain together multiple DMA reads from different parts of the flash memory. Typically, a CPU (in our case a MIPS clone) is in charge to control reconfiguration and to fill the size and address FIFOs with DMA commands. In other words, the configuration process is then a sequence of different DMA transfers which can be issued to a queue. In addition to DMA transfers, the value of '0' for the size is used to send a data word (stored in the start address Fifo) directly to ICAP, without initializing a DMA transfer from the flash memory. This is intended to add address information to a bitstream. Note that instead of determining addresses by module relocation of netlists and analyzing bitstreams, a driver might perform address computations at run-time.

3) Evaluation: In this paragraph, we present synthesis results and configuration speeds that were achieved by our configuration controller. As this controller is pure overhead, we optimized it for area. A resource breakdown is listed in Table I. As can be seen, without the UART (which would probably not be necessary for a commercial product), our controller needs only a bit more than 100 slices on a Spartan-6 FPGA (and no BRAM or DSP primitive).

The measured reconfiguration speed ranges from 73 MB/s to 97 MB/s, which mainly depends on the achieved compression ratio, which in turn, correlates with the logic/routing utilization inside the module bounding box. While this is slower than for example, [15] (400 MB/s), we can already provide up to 2.43 $\times$ the specified maximal ICAP reconfiguration speed of a Spartan-6 FPGA (40 MB/s) [12]. Moreover, our configuration controller takes considerable less resources (263/355 LUTs versus 586 LUTs and 8 BRAMs in [15]), provides hardware accelerated bitstream decompression and has built-in support

for module relocation. As a reference: the Xilinx HWICAP controller needs 684 LUTs and achieves less than 10 MB/s throughput when used together with a microblaze [12].

4) Discussion: Hardware supported bitstream relocation was firstly presented for the REPLICA project [20], [21]. In that work, one continuous partial bitstream is processed in a bitstream filter for updating the position field with displaced addresses. This approach costs more logic than adding distributed memory Fifos in front of the configuration controller (for storing a job queue). Furthermore, the parser has to be adapted for every FPGA and different filter logic is needed for each FPGA family (e.g., [20] for Virtex-E and [21] for Virtex-II). Moreover, the filtering works only for FPGAs where the bitstream encoding is homogenous, which is a restriction when using filtering for recent Xilinx devices, such as Spartan-6 FPGAs.

Our offline relocation and compression approach is more generic and does not relay on assumptions about the bitstream encoding and we also do not have to understand the commands inside the bitstream. This simplifies a reuse of our configuration controller in different systems. And if more advanced bitstream manipulations are needed, this can still be performed in software. This can still result in high-speed reconfiguration by transferring most of the configuration data per DMA while only adding small changes directly to the bitstream (e.g., modifying a position field or for changing a LUT function table).

V. Using EasyPR Remotely

The here presented system can be tried out remotely on the Blinded project website [22]. This project combines various FPGA labs for teaching VHDL design remotely via the internet. This includes basic setups with tasks such as state machine design and stepper motor control as well as advanced setups related to cryptography, CPU design, and partial reconfiguration.

For trying out EasyPR, registered users can upload their own video overlay modules and specify a bounding box. With this information, our tools will then run all synthesis steps (on the server) in order to create a relocatable partial bitstream. If this was successful, it is possible to upload the result (or other modules) to an Atlys Spartan-6 board. Loading and removing modules is controlled by the user in his web browser. For testing the result, a test video sequence is supplied as an HDMI stream to the board, then processed by the partial modules, and finally streamed back to the user. We are currently working on a video upload function for live video processing.

VI. Conclusion

With EasyPR, we provide a complete PR system including all design files as open-source (download: [10]). Design parameters have been carefully selected to allow engineers who are interested in PR a successful implementation of a reconfigurable system in only a few hours - or even within an hour when implementing partial modules for the static OpenPR reference system. With our new SPI flash DMA reconfiguration manager, we further help to provide fast bitstream transfer and easy module relocation. With this work, we hope to stimulate research on PR by making reconfigurable system design more accessible. Future work will include more supported boards, modules, and examples with more advanced PR features.

Acknowledgment

Removed for review.

References

[1] C. Claus, R. Ahmed, F. Altenried, and W. Stechele, "Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems," in *Proceedings of the 6th international conference on Reconfigurable Computing: architectures, Tools and Applications (ARC)*. Springer, 2010, pp. 55–67.

[2] E. J. McDonald, "Runtime FPGA Partial Reconfiguration," in *Proceedings of the 6th international conference on Reconfigurable Computing: architectures, Tools and Applications (ARC)*. Springer-Verlag, 2007, pp. 55–67.

[3] M. Feilen, M. Ihmig, C. Schwarzbauer, and W. Stechele, "An Efficient DVB-T2 Decoding Accelerator by Time-Multiplexing FPGA Resources," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 75–82.

[4] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium*, pp. 45–52, 2012.

[5] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011, pp. 228–235.

[6] Blind, "Removed for review," in *blind*, pp. 00–00.

[7] ——, *Removed for review*. Blind.

[8] Xilinx Inc., "Partial Reconfiguration User Guide (UG702)," Dec. 2010, rel 12.1.

[9] Altera Inc., "Quartus II Handbook (Volume 1: Design and Synthesis)," Dec. 2013, rel 13.0.

[10] Blind, "Removed for review," `http:\\blind...`

[11] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and Customization of FPGA-Based Soft Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 266–277, Feb. 2007.

[12] Xilinx Inc., "LogiCORE IP XPS HWICAP (v5.01a)," Jun. 2011, Design Guide DS586.

[13] S. Bayar, M. Tükel, and A. Yurdakul, "A self-reconfigurable platform for general purpose image processing systems on low-cost spartan-6 fpgas," in *ReCoSoC*, 2011, pp. 1–9.

[14] Shaoshan Liu and Richard Neil Pittman and Alessandro Forin, "Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller," Sep. 2009, Microsoft Tech. Report MSR-TR-2009-150.

[15] K. Vipin and S. Fahmy, "A high speed open source Controller for FPGA Partial Reconfiguration," in *Field-Programmable Technology (FPT), 2012 International Conference on*, 2012, pp. 61–66.

[16] C. Claus, F. H. Müller, J. Zeppenfeld, and W. Stechele, "A new Framework to Accelerate Virtex-II Pro Dynamic Partial Self-reconfiguration," in *Proceedings of the IEEE 21th International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, California, USA, March 2007, pp. 1–7.

[17] J. A. Storer and T. G. Szymanski, "Data Compression via Textual Substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[18] Blind, "Removed for review," *blind*, pp. 0–0.

[19] Z. Li and S. Hauck, "Configuration Compression for Virtex FPGAs," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 147–159.

[20] H. Kalte, G. Lee, M. Porrmann, and U. Rückert, "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop (IPDPS)*. IEEE Computer Society, 2005.

[21] H. Kalte and M. Porrmann, "REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs," in *Proceedings of the 3rd conference on computing frontiers (CF)*. New York, NY, USA: ACM, 2006, pp. 403–412.

[22] Blind, "Removed for review," `http:\\blind...`