Enabling Dynamic and Partial Reconfiguration in Xilinx SDSoC

Tobias Kalb, Diana Göhringer
Application-Specific Multi-Core Architectures (MCA) Group
Ruhr-University Bochum (RUB), Germany
{tobias.kalb, diana.goehringer}@rub.de

***Abstract*—In the past years dynamic partial reconfiguration (DPR) has been established as a well-known technique for systems featuring a field programmable gate array (FPGA). Systems-on-Chip (SoC) with an ARM processor ease the utilization of DPR and motivate its implementation to make use of the obvious advantages, such as the reduction of area, power and the acceleration of reconfiguring the FPGA. Nonetheless, the development process for SoCs is still a complex and time consuming task, especially for those designs using DPR. Xilinx counters this complexity with the introduction of their new high-level tools, namely the SDx Development Environment. The SDSoC Development Environment accelerates the development of designs running on Zynq 7000 devices by only using C/C++ applications as input. Unfortunately, this high-level workflow does not incorporate DPR. This paper shows an approach on how to use DPR in Xilinx SDSoC. Thus an application specific design can benefit from both the high-level workflow and the advantages of DPR. We show that our approach to DPR in SDSoC accelerates the overall design time and creates a more efficient embedded application. In our use case the dynamic and partial reconfiguration of hardware accelerators takes 10 ms and the hardware-related section of our embedded application is accelerated by a factor of 14 due to DPR.**

Keywords—dynamic partial reconfiguration, Xilinx, SDSoC, Zynq, Tcl, ZedBoard, FPGA

I. INTRODUCTION

Dynamic partial reconfiguration (DPR) is a common technique to design adaptive and flexible hardware on field programmable gate arrays (FPGAs). The main idea is to substitute defined regions of programmable logic containing reconfigurable modules at runtime. The partial reconfiguration process takes place without interfering with other parts of the system. Thus the functionality of a component of the system can be changed on demand [4] [5]. DPR provides several advantages.

First, swapping hardware accelerators in and out when needed offers a significant reduction of the required chip area. The re-use of hardware resources provides usability for the FPGA comparable to time-division multiplexing. This feature can also be described as a virtual extension of the chip area. Second, DPR leads to a reduction of power consumption. The described reduction of required chip area advertises the use of smaller FPGAs, which are not only cheaper but also offer lower power consumption. The overall power consumption may be reduced further by using DPR to unload system components completely if they are redundant [6]. Finally, considering the timing properties of a designed SoC, reconfiguring the complete FPGA provides basic runtime adaptivity but may, however, take too long, e.g. for hard real-time constraints. Partial reconfiguration, in contrast, accelerates this process of updating system components by only having to upload partial configurations. The features of DPR are very beneficial for the design of embedded systems. Especially SoCs like the Xilinx Zynq 7000 device family can profit from DPR [10]. The possibility for an operating system running on the ARM Cortex-A9 eases the partial reconfiguration process. This allows for software-controlled DPR using the Processor Configuration Access Port (PCAP) interface [11]. Nevertheless, the development of SoCs with interdependent software and hardware components is a complex and time consuming procedure. Furthermore, the design of partially reconfigurable hardware is a complex task and still has to be done manually in a cascade of design steps by the system designer. DPR adds to the overall complexity and hence is mainly used by experts.

The newly introduced Xilinx SDSoC Development Environment for Zynq 7000 devices pursues the approach to manage the design complexity internally and to increase the level of abstraction for the system designer [8]. For this high-level workflow only C/C++ source code is needed as an input for the creation of hardware-accelerated heterogeneous SoCs. Xilinx SDSoC significantly accelerates the development of embedded applications which incorporate hardware accelerators. Unfortunately, this accelerated high-level approach does not include DPR so far. It means in effect that until today DPR is not supported using a high-level workflow. The workflow for most parts of the system is accelerated with modern development tools, yet the low-level approach to DPR decelerates the overall design process.

In this paper we present an approach to DPR as an integrated part of the high-level Xilinx SDSoC workflow. With a set of reusable Tcl scripts an embedded C/C++ application can be designed to use dynamically and partially reconfigurable hardware accelerators. The fundamental properties of an application can be debugged, analyzed and tested using SDSoC, before the final design step enables this application to exploit the benefits DPR. The timing properties for reconfiguring the programmable logic improve by a factor of 23 for the reconfiguration process and by a factor of 14 for

978-1-5090-3707-0/16/$31.00 ©2016 IEEE

the combined reconfiguration and execution of the hardware accelerators within the embedded application.

The remainder of the paper is structured as follows: In Section II an overview of related work in the area of DPR methodologies and development tools is provided. Section III describes the basics of the new Xilinx SDSoC Development Environment, which are important for this work. Section IV follows with the detailed description of the implemented Tcl extensions for DPR. The experimental results are presented in Section V, and in the final Section VI the paper finishes with a conclusion and further outlook.

II. Related Work

Since the introduction of DPR for FPGAs there has been a lot of research on how to make DPR hardware design easier. Oftentimes this results in specialized frameworks, object-oriented approaches and other solutions impeding reusability of the leveraged design process for DPR-enabled hardware for other designs and SoCs. In [1] an object-oriented framework for DPR is implemented. The high-level language POF (Parallel Object Language) is introduced, which is similar to Java. The object-oriented framework consists of a software-to-hardware compiler, as all POF source files are translated to VHDL. These VHDL files are then used within Xilinx PlanAhead to implement reconfigurable modules as part of a specialized hardware system running on a Xilinx Virtex device. A framework for automated high-level design is presented in [2]. The introduced framework PaRAT (partial reconfiguration amenability test) parses, analyzes and partitions an application written in a HLS language. Using Vivado HLS the framework is able to generate DPR hardware designs. In [3] the authors present a high-level framework called OSSS+R for reconfigurable systems. The framework uses SystemC for simulation and synthesis. The generated VHDL source files represent the input for Xilinx ISE to create the final bitstreams. In contrast to the aforementioned publications, our approach to DPR in SDSoC works without a specialized framework. Our extensions are designed to seamlessly integrate into the high-level workflow provided by the Xilinx SDSoC development environment. The presented Tcl scripts ensure flexibility and reusability. Additionally, our extension of SDSoC is targeted towards modern Xilinx Zynq 7000 devices, so that modern yet complex SoCs are able to use the advantages of DPR.

Creating a SoC, the system designer not only has to take care of the integration of DPR into the hardware design of the SoC. Furthermore, the usability of the DPR-enabled hardware has to be taken into account. In [4] the authors use a Linux operating system to partially reconfigure the FPGA of a Xilinx Zynq device. The usage of DPR is made easier, but still it is necessary to manually and separately design the software application and the hardware system. Another approach is presented in [5]. Here, a framework for partially reconfigurable hardware accelerators is presented. The accelerators are connected to the ARM processor of a Zynq Z-7020 device. A Linux operating system is running on the ARM processor. Specialized device drivers have been developed for using the hardware accelerators. Again, in contrast to the presented works, our approach to DPR is integrated into a comprehensive high-level workflow. In the presented publications, all the system components had to be developed separately. Thus a lot of effort has to be put into the development and implementation of the hardware/software-partitioned embedded application and the corresponding hardware design. Our Tcl scripts allow designing DPR-enabled hardware which is compatible to the standard SDSoC development process. Thus we are able to use the operating systems and device drivers created by SDSoC. Our Tcl scripts provide an efficient high-level extension of the SDSoC development environment.

III. Xilinx SDSoC

The Xilinx SDSoC Development Environment, introduced in July 2015, is a new development tool for Zynq 7000 devices targeted towards a high-level workflow for embedded hardware-accelerated software. The main input is a C/C++ application, in which one or several software functions can be marked for automated acceleration on a FPGA. To direct the generation of the hardware system, the designer can insert pragmas into the source code, thus defining constraints for the accelerated functions. A comprehensive introduction to the Xilinx SDSoC Development Environment is provided in [8].

A. Basic Workflow

At the beginning of the design the developer is able to select different development boards featuring a Zynq 7000 device. Additionally, an operating system can be chosen. Xilinx SDSoC offers bare-metal environments, a Linux and a FreeRTOS real-time operating system [7]. The device drivers and other necessary components - for the operating system and for the communication between processing system and programmable logic - are then generated automatically by the development environment. These features are realized by the full-system optimizing compiler, more precisely SDSCC for applications written in C language and SDS++ for C++ source code [8]. The combination of the two is further referenced as SDS compiler. A detailed description of the features of both the development environment and compiler is provided in [7].

B. High Level Approach

The core element of SDSoC that is responsible for achieving the high level of abstraction is the SDS compiler. The compiler automatically generates all necessary parts of the system, including a bitstream for the programmable logic and an executable application together with the selected operating system. After an initial analysis of the source code, the embedded application is partitioned into its hardware and software components. The generation of hardware accelerators relies on the Vivado HLS tool, which is invoked in the background of SDSoC. The linking of the individual system components is done by SDS++. This compiler further invokes the Vivado design tools to automatically integrate the hardware accelerators into a complete hardware design, including the communication infrastructure and data movers between processing system and programmable logic. In a final step SDSoC starts the Bootgen tool to create a bootable image containing the executable application for the processor and the bitstream for the FPGA [8].

C. Partitions

If an application is targeted towards a small device or the accelerated functions require a lot of chip area to meet timing constraints, SDSoC offers the possibility to create so called partitions for independent hardware accelerators. This option allows to time-multiplex the chip area for different hardware-accelerated software functions, as each hardware accelerator is realized on a separate full bitstream [7]. The SDSoC compiler then alters the executable application so that the programmable logic is configured with the correct full bitstream at runtime.

Using this technique within an accelerated embedded application introduces some benefits similar to those of DPR. The required chip area can be reduced and accordingly the power consumption decreases. However, the reconfiguration of the complete programmable logic has two major drawbacks. First, the full reconfiguration process takes much longer than a partial reconfiguration process. This means that for critical applications the timing constraints may not be met. Second, the programmable logic is updated entirely during runtime. SDSoC is capable of using custom hardware setups instead of the predefined Zynq 7000 development boards [9]. If a designer considers using SDSoC to hardware-accelerate software functions within a custom hardware setup several independent full bitstreams may not be sufficient. System functions on programmable logic, which are independent from the accelerated software functions, are unnecessarily interrupted, reloaded and restarted every time the full bitstream is updated. This may lead to loss of data, malfunction of attached sensors and actors and thus impedes the use of several partitions. However, this is one of the features where DPR exceeds as no surrounding system components are interrupted during the reconfiguration process of a partial region of the programmable logic.

IV. Implementation of DPR

As described in Section III, the core element of the SDSoC development environment is the full-system optimizing compiler SDSCC/SDS++. According to [7], the SDS compiler can either be run without options in standard mode, or the execution can be controlled by defining miscellaneous compiler and linker options.

A. SDSoC Linker Option

Since SDSoC does not support an own high-level approach for DPR, and hardware access is otherwise restricted to pragmas within the development tool itself, our implementation exploits the linker option *impl-tcl* to achieve high-level access to the creation of the hardware system. This option directs the SDS++ linker to invoke the Vivado design tools with a custom Tcl script instead of the automatically generated generic Tcl script. According to this approach we are able to combine all the convenience provided by the high level of abstraction of the SDSoC development environment with the advantages of a hardware design customized for DPR.

The design process of SDSoC runs unchanged for the analysis, optimization and compilation of the embedded application. Additionally, one or several compatible hardware accelerators can be inserted into the application. The custom Tcl script is called as soon as the Vivado tools are invoked for the creation of the final hardware design. Our implementation utilizes the possibilities given by the fact that the Vivado tools are entirely controllable by Tcl commands [12]. The design process of our Tcl scripts builds upon the basic hardware design provided by SDSoC for the particular embedded application. This ensures that our custom hardware is compatible with the software components generated by SDSoC. The following detailed steps of the custom Tcl scripts are entirely executed within the batch mode of Vivado.

B. Extending Tcl Scripts

The first task of our Tcl scripts is to create a logical and consistent folder structure for our DPR design. As Vivado is used in *non-project mode*, this structure ensures convenient data management for all design files generated for the DPR hardware system. However, in the first section of the scripts we set necessary variables to distinguish between different operating systems. Thus our extension of the SDSoC development environment can be used on Microsoft Windows as well as on Linux-based operating systems. Following this, synthesis is run for the basic SDSoC hardware design to create an initial checkpoint for the DPR design. The Vivado design suite uses checkpoints to save the current state of any design.

The second set of related tasks creates all hardware accelerators, which are intended to be used as dynamically and partially reconfigurable modules. Here, the user is able to add, remove or change hardware accelerators. Synthesis for these IP cores is then executed in out-of-context mode [10]. All checkpoints for synthesized hardware accelerators are gathered in the corresponding folder of our folder structure. These checkpoints are used in the following progress of the hardware design, as they represent the reconfigurable modules of the overall design. Since our Tcl scripts are designed for reusability, these steps easily can be edited and run for any desired number of compatible hardware accelerators.

With the basic checkpoints for the hardware accelerators generated, our scripts then perform the third set of tasks. This set is crucial for hardware designs incorporating DPR. The accelerators are defined as reconfigurable modules and the floorplanning for all reconfigurable modules is executed. The floorplanning is an essential design step for DPR, as it defines the region within the programmable logic a partial bitstream - the final product of a reconfigurable module - is loaded to. Each reconfigurable module requires a defined partition and all functions assigned to a specific have to be compatible in terms of input and output ports. As soon as the floorplanning is finished, the initial design is implemented. In our implementation the initial floorplanning has to be completed manually. This includes analyzing the initial design and providing a hard-coded floorplan considering all reconfigurable modules in advance. In future work we plan to further leverage our extension of SDSoC to execute an automated floorplanning for the reconfigurable hardware accelerators.

The fourth and penultimate set of Tcl scripts processes the designs steps for the actual DPR. At first, with the checkpoint of the first full design opened, all hardware accelerators, or more precisely all present reconfigurable modules, are

unloaded and updated as black boxes. The remaining design represents the static part of the hardware, which is stored as an individual checkpoint and results in a bitstream without any reconfigurable hardware accelerators, only containing the static communication interconnections. The static part of the design includes all non-reconfigurable functions as well as all interconnects to the reconfigurable modules, i.e. AXI interconnects. This checkpoint is subsequently used to load every reconfigurable module into every reconfigurable region. This procedure ensures providing checkpoints for all required full and partial hardware designs. After all, the fifth and last part issues the commands for generating all full and partial bitstreams. Fig. 1 shows the extension of SDSoC for DPR with our implemented Tcl scripts.

C. Reintegration into SDSoC

Finally, the custom Tcl scripts then seamlessly reintegrate from the hardware design run for DPR into the standard SDSoC design process. The SDSoC development environment finishes its standard procedure and creates a bootable SD card. All full and partial bitstreams are ready to use with the hardware-accelerated embedded application created using the high-level SDSoC workflow. Section V continues with a detailed example.

V. Experimental Results

To validate our extension of SDSoC, we developed a hardware/software-partitioned application with two software functions realized on the programmable logic of the Xilinx Z-7020 device featured on the ZedBoard. The functions selected for implementation on the FPGA are kept simple, as the primary focus is on the demonstration of DPR as a part of the high-level workflow of SDSoC. The first function executes a subtraction of two matrices, while the second function realizes an addition of the same two matrices.

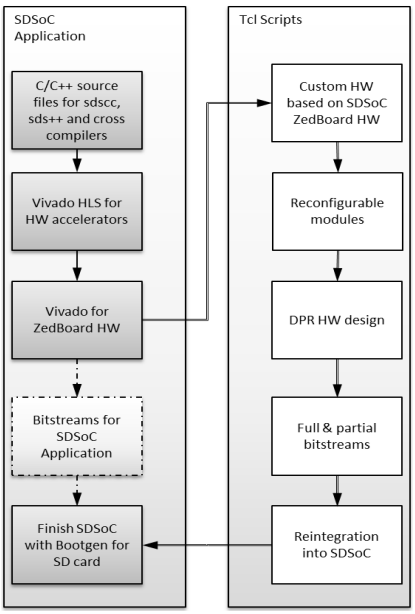

Fig. 1. Extension of SDSoC for DPR using Tcl

A. Concept of Test Application

For our test application we set up an SDSoC application project, which uses a Linux operating system running on the ZedBoard. The clock frequencies of the FPGA are set to 142.86 MHz, as these frequencies are automatically suggested by SDSoC. The developed C++ application is similar to the SDSoC example project executing a matrix multiplication and addition. However, using an UART terminal to control the test application, we decided to use a matrix subtraction and addition, as the results of the computation can be checked faster and easier. Thus the verification of a correct dynamic and partial reconfiguration is ensured. The procedure of the test application consists of the C++ main function calling a testing sequence. The testing sequence *maddsub_test* then again executes the two functions chosen to be implemented on the FPGA, i.e. the matrix subtraction *msub* and the matrix addition *madd*. Both these functions are described only using C++ source files.

The testing function *maddsub_test* first initializes all required matrices with their dimensions set to 32x32. Here, we require a total number of four matrices, namely *A*, *B*, *tmp1* and *tmp2*. While *tmp1* and *tmp2* are used to store the results received from the FPGA, the matrices *A* and *B* are fed into the hardware accelerators. The elements of the matrices *A* and *B* are both initialized consecutively according to their index. That means that the first row stores the values 0 to 31, the second row stores the values 32 to 63 and so forth. The expected result for the subtraction then is the zero matrix, while the addition results in the doubling of the value of each matrix element. These functions have been chosen for a convenient verification of the hardware accelerators and the DPR. With all matrices initialized, the software application finally calls the C++ functions *msub* and *madd* for the calculations realized as hardware accelerators. The execution of the functions is measured using a timer for processor cycles. The executions of the matrix calculations are included in a loop, together with the timing measurement and the function calls for reconfiguration of the FPGA. This loop runs for 16 iterations to provide average values for the timing information about execution time of the hardware accelerators and about full or partial reconfiguration of the programmable logic. It is vital for DPR to provide compatible hardware designs. This implies that the software functions selected in SDSoC for hardware acceleration have to be compatible to each other, ensuring interchangeable partial bitstreams. In SDSoC this can be achieved by a) using the same names for input and output parameters of the software functions selected for acceleration, and b) by inserting SDS pragmas defining the data movers and the data access patterns. Fig. 2 shows the block design of the hardware accelerators. The ports feature the same names and implementation details and thus are interchangeable. Also, SDSoC creates device drivers for all hardware accelerators of an SDSoC application, so that the executable binary file is able to call the hardware accelerators. With the hardware accelerators being compatible and interchangeable, the drivers are also compatible and interchangeable. This is an important detail for DPR, as the software application running on the ARM processor now is able to use different hardware accelerators on the programmable logic with one and the same function call.

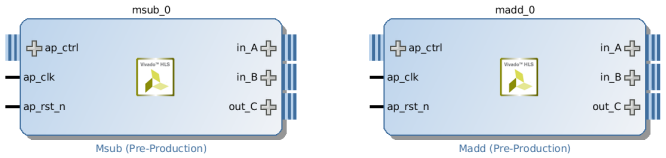

Fig. 2. Block design of hardware accelerators msub and madd

B. *Versions of Test Application*

With the basic functionality of our test application being the same, we implemented several versions. The first version *appBase* uses the standard SDSoC workflow to port both software functions *msub* and *madd* onto the FPGA. Here, both functions are realized as hardware accelerators on the programmable logic at the same time. The output of this first version is a single complete bitstream without the possibility of full or partial reconfiguration. This design is used for testing the hardware accelerators and for setting a baseline regarding area and power requirements. The second version *appPartition* exploits the SDS pragma *partition(id)* for creating several partitions. Here, the hardware accelerators *msub* and *madd* are implemented on separate full bitstreams. As explained above, the full bitstreams are then reconfigured 16 times to provide average values about the execution time of the application. We use the second version of the test application to compare the reconfiguration of full bitstreams and partial bitstreams.

The third version *appDPR* is run with the additional SDS++ linker option *impl-tcl* and invokes our custom Tcl scripts for the creation of a DPR hardware design within the high-level workflow of SDSoC. The custom hardware design is compatible with the software application compiled by SDSoC. Furthermore, to demonstrate our approach to DPR within the SDSoC development environment, the software application only calls one function, i.e. one driver, to execute the matrix calculations on the FPGA. This proves that our Tcl scripts create compatible hardware, as the same function call executed on the ARM processor receives different results from the programmable logic. With the third version *appDPR* we show that our extension of SDSoC creates DPR-enabled hardware designs integrated into the high-level workflow of SDSoC. An embedded application with one or several hardware accelerators can be tested and evaluated using the standard procedure of the SDSoC development environment. In a final step our Tcl scripts are invoked by the SDS++ linker to create compatible hardware designs featuring one or several dynamically and partially reconfigurable hardware accelerators. The three different versions of the test application are shown in Fig. 3.

C. *Evaluation*

With the functionality of our extension of SDSoC verified by the test application *appDPR*, we use the output of all three versions to evaluate the area and power requirements. The latter two versions of the application *appPartition* and *appDPR* are then used to demonstrate the advantages of DPR regarding the execution time of an embedded application incorporating DPR. Finally, we are going to show that our extension for DPR within SDSoC leads to an efficient embedded application combining both the advantages of the high-level workflow of SDSoC with the advantages of DPR.

1) Area Requirements

Regarding the area requirements on programmable logic our extension of SDSoC shows that DPR reduces the required chip area even with small accelerators. TABLE II depicts the absolute numbers of needed resources for both hardware accelerators *msub* and *madd*. As these functions are small and used for testing DPR, their area requirements are determined by the dimensions of the matrices. The relative sizes of the accelerators are based on the available resources of the Z-7020 device featured on the ZedBoard.

TABLE I compares the full bitstreams of the basic test application *appBase* featuring two hardware accelerators with both full bitstreams of the application *appPartition*. Here, each bitstream features one hardware accelerator realized on programmable logic. Furthermore, these three different hardware designs are compared to the DPR-enabled bitstreams of the test application *appDPR*. As expected, the full bitstreams of *appPartition* and *appDPR* - each including one hardware accelerator on the FPGA - provided effectively identical results. The deviation measures 14 and 15 look-up tables (LUTs) in total, which corresponds to 0.03 percent of the available number of LUTs of the ZedBoard. The possibility to use the static design decreases the amount of required LUTs by a further number of 256. The table shows that the area requirements are decreased by a hardware design incorporating DPR, although in our demonstration the savings are – because of the rather small hardware accelerators – not that significant. For future work different hardware accelerators are planned, providing more significant results regarding area requirements.

2) Power Requirements

For the evaluation of the power consumption we use the *report_power* command of Vivado. The power estimations are executed with the default settings of the Vivado design tools. Unfortunately, the power estimation does not work for hardware designs featuring reconfigurable regions or black boxes. Thus the estimation for the DPR designs of *appDPR* cannot be provided.

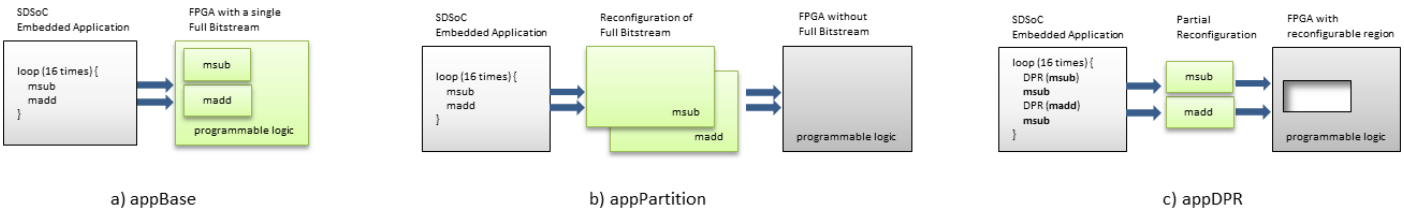

Fig. 3. Applications for testing the extension of SDSoC for DPR

TABLE I. RESOURCES OF COMPLETE HARDWARE DESIGNS

Type	Total	appDPR						appPartition				appBase	
		static		*msubb*		*madd*		*msubb*		*madd*		*complete*	
		Used	*Util %*	*Used*	*Util %*	*Used*	*Util %*	*Used*	*Util %*	*Used*	*Util %*	*Used*	*Util %*
Slice LUTs	53200	7046	13.24	7288	13.70	7288	13.70	7302	13.73	7303	13.73	8147	15.31
LUT as Logic	53200	6409	12.05	6643	12.49	6642	12.48	6656	12.51	6657	12.51	7468	14.04
LUT as Memory	17400	637	3.66	645	3.71	646	3.71	646	3.71	646	3.71	679	3.90
Slice Registers	106400	11270	10.59	11694	10.99	11694	10.99	11694	10.99	11694	10.99	13531	12.72
Register as Flip Flop	106400	11270	10.59	11694	10.99	11694	10.99	11694	10.99	11694	10.99	13531	12.72
Register as Latch	106400	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00

TABLE II. RESOURCES OF HARDWARE ACCELERATORS

Type	Total	msub		madd	
		Used	*Util %*	*Used*	*Util %*
Slice LUTs	53200	242	0.45	242	0.45
LUT as Logic	53200	234	0.44	233	0.44
LUT as Memory	17400	8	0.05	9	0.05
Slice Registers	106400	424	0.40	424	0.40
Register as Flip Flop	106400	424	0.40	424	0.40
Register as Latch	106400	0	0.00	0	0.00

TABLE III. POWER ESTIMATIONS

Power [mW]	appPartition		appBase
	msub	*madd*	*complete*
Total On-Chip Power	1798	1801	1823
Dynamic	1637	1639	1661
msub	4	-	4
madd	-	3	3

TABLE IV. TIMING MEASUREMENTS USING PROCESSOR CYCLES

	appPartition	appDPR
	SDSoC Partitions	*Extension for DPR*
Number of Reconfigurations	32	32
Avg. Processor Cycles per Reconfiguration	446,036,742	31,322,161
Speedup	1	14,24

Therefore, as a future work, we plan to further evaluate the power requirements using the ZC-702 development board. This development board allows for measuring the power consumption for both the ARM processor and the FPGA of the Zynq device at runtime. TABLE III lists the estimated power consumptions for *appBase* and *appPartition*. The hardware accelerators *msub* and *madd* feature an estimated power consumption of 3 mW and 4 mW respectively. By comparing *appBase* to *appPartition*, it can be seen that the implementation of only one hardware accelerator at a time results in an estimated reduction of 22 mW and 25 mW of total power consumption. Compared to a single hardware accelerator, the increased power reduction again can be accounted to the reduction of communication infrastructure required for driving the hardware accelerators. Unfortunately, separate power estimations for *appDPR* cannot be provided at this stage, but, considering that the hardware designs are fully compatible and the area requirements are identical to the hardware design of *appPartition* the results are expected to be similar without a lot of deviation.

3) Duration of Reconfiguration

Finally, we evaluate the timing properties of our extension of the SDSoC development environment. Here, we compare the embedded applications *appPartition* and *appDPR*. As explained above, *appPartition* is designed to implement both hardware accelerators *msub* and *madd* within separate full bitstreams. The advantage is that the designer only has to take care of the high-level C/C++ application.

However, the application *appDPR* uses our extending Tcl scripts to accomplish the same behavior as *appPartition* with partial bitstreams used for the reconfiguration process instead of full bitstreams. We mimic the modification of the application by hard-coding system calls into the source code. At runtime these system calls are handed to the Linux operating system and execute the dynamic and partial reconfiguration of the FPGA. The timing behavior of the reconfiguration process is measured by issuing system calls for full and partial reconfiguration with the preceded *time* command in for the Linux shell. The execution times measured for full and partial reconfiguration are depicted in Fig. 4. Here, the measurements for *sys* represent the time the kernel spends for the execution of the reconfiguration process. The reconfiguration of a full bitstream takes at least 230 ms, while the partial reconfiguration of both hardware accelerators is accomplished within 10 ms of kernel time. The sizes of the partial bitstreams are 196.7 kB each. The measurements for *user* amounts to 0 ms for all bitstreams. This shows that the reconfiguration of a partial bitstream is 23 to 24 times faster than the reconfiguration of a full bitstream.

As described above, the test applications call both hardware accelerators within a loop for a total number of 16 runs. These timing measurements include the reconfiguration procedures and execution of the hardware accelerators for both the applications *appPartition* and *appDPR*. When the loop is finished the average number of processor cycles spent for reconfiguration and execution is calculated. Using this method for the evaluation of our extension provides a better understanding of DPR as part of a running embedded application, as a total number of 32 reconfigurations take place at runtime. The results are depicted in TABLE IV. These average timing values demonstrate the real advantage of the extension of SDSoC for DPR. Not only is one single partial reconfiguration at least 23 times faster than the reconfiguration of a full bitstream. Moreover, the reconfiguration and subsequent execution of hardware accelerators of the embedded application is accelerated by more than 14 times.

This shows that our integration of DPR into the SDSoC development environment combines the advantages of DPR with the faster development process of the high-level workflow of SDSoC.

VI. Conclusion and Future Work

In this paper we presented an extension of the new Xilinx SDSoC Development Environment. We implemented a set of Tcl scripts for creating hardware designs for DPR. The Tcl scripts seamlessly integrate into the SDSoC tool flow, as they are invoked exploiting the corresponding SDS++ linker option. We then showed that our compatible hardware designs improve the resulting SoC with regard to area and power requirements. The most significant improvements have been gained in view of the execution time of an embedded application. Our Tcl scripts were used to easily create an embedded application with SDSoC which executes several partial reconfigurations at runtime. This shows that not only the time spent designing a DPR-enabled SoC is accelerated, but also the execution of the embedded application running on that SoC is accelerated by a notable factor.

For future work we plan to further evaluate our extension using computationally more expensive software functions selected for hardware acceleration. These designs can then be used to confirm our results regarding area and power requirements. As our extending Tcl scripts have been designed for reusability, we also plan to evaluate our extension with more complex SoCs using relocation and automated floorplanning. Real-time features will be evaluated using the FreeRTOS operating system instead of a Linux operating system. Thus we are going to further investigate the advantages provided by the combination of SDSoC and DPR.

Acknowledgment

The work is funded by European Commission under the H2020 Framework Programme for Research and Innovation under grant agreement No 688403.

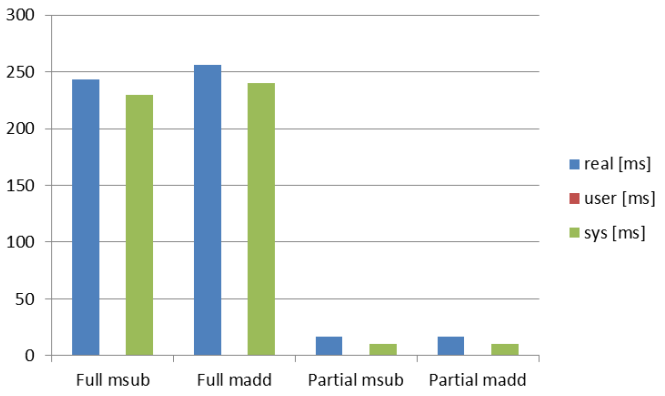

Fig. 4. Timing measurements for reconfiguration within Linux

References

[1] Abel, N., "Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration", International Conference on Field Programmable Logic and Applications (FPL), p. 240 – 243, Sept. 2010

[2] Kumar, R., Gordon-Ross, A., "An Automated High-Level Design Framework for Partially Reconfigurable FPGAs", IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), p. 170 – 175, May 2015

[3] Schallenberg, A., Nebel, W., Herrholz, A., Hartmann, P.A., Oppenheimer, F., "OSSS+R: A framework for application level modelling and synthesis of reconfigurable systems", Design, Automation & Test in Europe Conference & Exhibition (DATE '09), p. 970 – 975, April 2009

[4] Al Kadi, M., Rudolph, P., Göhringer, D., Hübner, M., " Dynamic and partial reconfiguration of Zynq 7000 under Linux", International Conference on Reconfigurable Computing and FPGAs (ReConFig), p. 1 – 5, Dec. 2013

[5] Prince, A. A., Kartha, V., "A framework for remote and adaptive partial reconfiguration of SoC based data acquisition systems under Linux", 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), p 1 – 5, July 2015

[6] W. Lie, Feng-yan, W., " Dynamic Partial Reconfiguration in FPGAs", Third International Symposium on Intelligent Information Technology Application (IITA), p. 445 – 448, Nov. 2009

[7] Xilinx Inc., "SDSoC Environment User Guide (UG1027)", http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1027-sdsoc-user-guide.pdf, Accessed on 14.03.2016

[8] Xilinx Inc., "SDSoC Environment User Guide: Getting Started (UG1028)",http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1028-intro-to-sdsoc.pdf, Accessed on 14.03.2016

[9] Xilinx Inc., "SDSoC Environment UserGuide: Platforms and Libraries (UG1146)",http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1146-sdsoc-platforms-and-libraries.pdf, Accessed on 14.03.2016

[10] Xilinx Inc., "Vivado Design Suite User Guide: Partial Reconfiguration (UG909)",http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug909-vivado-partial-reconfiguration.pdf, Accessed on 14.03.2016

[11] Xilinx Inc., "Zynq-7000 All Programmable SoC: Technical Reference Manual(UG585)",http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, Accessed on 14.03.2016

[12] Xilinx Inc., "Vivado Design Suite Tcl Command Reference Guide (UG835)",http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug835-vivado-tcl-commands.pdf, Accessed on 14.03.2016