PARAMETERIZED AND ENERGY EFFICIENT ADAPTIVE BEAMFORMING ON FPGAS USING MATLAB/SIMULINK *

Jingzhao Ou, Viktor K. Prasanna

Electrical Engineering - Systems
University of Southern California, Los Angeles, CA 90089
Emails: {ouj,prasanna}@usc.edu

ABSTRACT

Adaptive beamforming is widely used in many sonar and telecommunication systems. FPGAs are attractive for implementing these applications. In this paper, we develop parameterized designs and identify energy efficient FPGA implementations for adaptive beamforming applications using MATLAB/Simulink based system-level design tools. Experimental results are given to show that up to 51% energy reduction can be achieved using our design approach.

1. INTRODUCTION

An *adaptive beamforming system* combines the signals received from multiple antenna elements and forms pencil beams adaptively in response to the signal environment so as to receive a source signal radiating from a specific direction and to attenuate signals originating from other directions that is of no interest. One important use of adaptive beamforming systems is in the area of software defined radio (SDR) [4], where they are used to improve system capacities of the base stations. Adaptive beamforming systems demand multiple orders of magnitude data processing abilities than single-antenna systems, in which adaptive beamforming is not employed. Different wireless communication protocols require different implementations of the adaptive beamforming systems. Thus, parameterized hardware designs of adaptive beamforming are desired. Moreover, many wireless systems using SDR are energy constrained while the high data processing requirement of adaptive beamforming consumes much energy.

Large densities of recent FPGAs have enabled the development of complex digital signal processing applications using them. Especially, FPGAs are an attractive option for implementing adaptive beamforming in SDR due to their high performance, low power dissipation per unit computation, and reconfigurability [4].

MATLAB/Simulink based design tools, such as *DSP Builder* [1] from Altera and *System Generator* [8] from Xilinx, are becoming popular for developing digital signal processing applications on FPGAs. In *System Generator*, IP (Intellectual Property) cores and HDL (Hardware Description Language) designs are made available through the block set within *System Generator*. The user assembles a design by using the blocks from the block set and connecting them via a GUI. After the design is completed, it is automatically translated into the corresponding HDL implementation. Testbenches are also automatically generated for hardware simulation and verification.

There are two major advantages offered by these tools. One is that there is no need to know HDLs. This allows researchers and users from the signal processing community, who are usually familiar with MATLAB/Simulink and unfamiliar with HDL, to get involved in the hardware design process. The other advantage is that the designer can make use of the ability of MATLAB/Simulink to perform arithmetic-level simulation, which is much faster than behavioral and architectural simulations in traditional FPGA design flows.

However, there are several limitations in using the current MATLAB/Simulink design flow to develop adaptive beamforming applications as more emphasis is placed on the energy efficiency of their implementations. One is that the current versions of the tools have no support for rapid energy estimation of FPGA based designs. Due to lack of a high-level architecture as that of general purpose processors, energy estimation of FPGA designs requires techniques such as domain-specific modeling proposed in [2]. Accurate estimation using RTL (Register Transfer Level) simulation is too time consuming and can be overwhelming considering the fact that there are usually many possible implementations of an application on FPGAs. Another limitation is that these tools do not provide an interface for traversing the MATLAB/Simulink design space and help the users to identify energy efficient designs while the design requirements are satisfied.

We have developed an add-on tool of MATLAB/Simulink, called *PyGen*, the architecture of which is discussed in [6]. Using this tool, MATLAB/Simulink designs can be described using Python scripting language [7]. The contribution of this paper is in *the use of PyGen to develop parameterized MATLAB/Simulink designs and identify the designs that lead to energy efficient implementations of adaptive beamforming applications on FPGAs*. While our approach can be applied to any

*SUPPORTED BY THE DARPA POWER AWARE COMPUTING AND COMMUNICATION PROGRAM UNDER CONTRACT NO. F33615-C-00-1633.

MATLAB/Simulink based design tool, *System Generator* [8] is used due to tool availability.

The remainder of this paper is organized as follows. Section 2 discusses our design flow using *System Generator* and *PyGen*. Section 3 presents the development of an MVDR (Minimum Variance Distortionless Response) adaptive beamforming application using this design flow. The performance of various designs is also given in this section. Finally, we conclude in Section 4.

2. A MATLAB/SIMULINK BASED DESIGN FLOW

Our design flow is shown in Figure 1. The shaded blocks correspond to the four major additional functionalities provided by *PyGen* to the original *System Generator* design flow. These are: (1) support for development of parameterized designs; (2) rapid energy estimation using techniques in [2]; (3) profile of energy dissipation of a design; (4) optimization of designs for energy efficiency.

We begin by describing our designs either in *PyGen* or directly in the GUI provided by *System Generator*. Parameterized designs are developed in the form of Python classes. By instantiating these classes and generating Python objects with the desired parameters, *PyGen* automatically creates the corresponding *System Generator* designs. For example, instantiating a Python class `Mult()` with `n_bits=18` has the same effect as dragging a multiplication block into the design and setting its number of bits to 18 with the other parameters taking their default values. A complete design usually contains a hierarchy of such Python objects. Using the arithmetic simulation ability of MATLAB/Simulink, we can quickly debug and verify the correctness of our designs.

A performance model is associated with each generated Python object. Input to these performance models are their settings in the design and test data from the previous Simulink simulation. Design constraints, such as latency, throughput, and available hardware resources (number of slices, dedicated multipliers and Block RAMs for Xilinx FPGAs), are also described in *PyGen*. Then, based on the performance models and the design constraints, *PyGen* performs optimization by automatically instantiating the Python classes with different parameters, estimating and comparing their performance, and ensuring that the design constraints are satisfied. Finally, using *System Generator*, an HDL implementation of the final design that achieves maximum energy efficiency is generated. By going through the usual FPGA synthesis and place-and-route process, this HDL implementation can be translated into bitstreams and downloaded to the device.

The design flow can be iterative. Relying on tools such as *XPower* [8], *PyGen* can analyze the results from low-level simulation and generate the profile of energy dissipation of a design. Such profiling results can be used to improve the accuracy of the performance estimation.

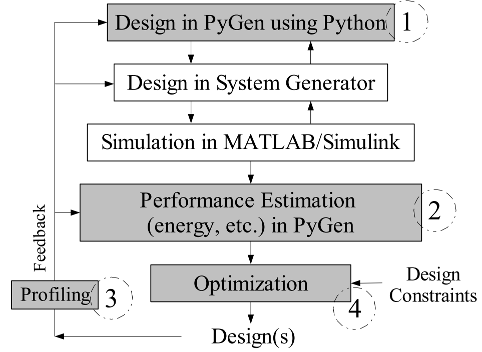

Fig. 1. Design Flow

3. DESIGN OF AN MVDR ADAPTIVE BEAMFORMING APPLICATION

3.1. MVDR Beamforming

We consider an LMS (Least Mean Squares) based MVDR adaptive beamforming application studied in [5]. It consists of two tasks: *filtering* which performs the calculation specified in Equation (1), and *updating of the coefficients of the filter*, which performs the calculation specified in Equation (2).

$$y(n) = \underline{W}^H(n') \cdot \underline{X}(n) \tag{1}$$

$$\underline{W}(n'+1) = \underline{W}(n') - \mu \cdot y^*(n) \cdot \left(\underline{X}(n) - \frac{\underline{U}^H \cdot \underline{X}(n) \cdot \underline{U}}{\underline{U}^H \cdot \underline{U}} \right) \tag{2}$$

where, $\underline{X}(n)$ is a vector that represents the input data samples, $y(n)$ is the filter output, $\underline{W}(n')$ is a vector that represents the coefficients of the filter, μ is the step-size parameter, and $\underline{U}$ is the steering vector. The size of the vectors is decided by the number of antenna elements employed by the antenna system. Updating of the filter coefficients are not required for each set of input data. The filtering task and the updating task can run at different speeds. Also, we assume that the beamforming is performed before modulation and thus uses complex number operations.

3.2. Parameterized Designs

The parameterized design of the beamforming application is implemented as a Python class `MVDR()`. The input parameters to this class are decided by the beamforming algorithm, the hardware architectures used, and the design requirements. They are `spdFiltering` (operating speed of filtering), `spdUpdate` (operating speed of coefficient updating), `n_bits` (number of bits), `bin_pt` (binary point position), `degParFiltering` (degree of parallelism for filtering), `degParUpdate` (degree of parallelism for coefficient updating), `dedMultFiltering` (number of dedicated multipliers used in filtering), `dedMultUpdate`

(number of dedicated multipliers used in coefficient updating), `sVec` (vector size), `mu` (step size μ), `U` (steering vector).

The `MVDR()` class contains two Python classes, `Filtering()` and `Updating()`, which implement the filtering task and the coefficient updating task, respectively. Input to class `Filtering()` is: `spdFiltering`, `n_bits`, `bin_pt`, `sVec`, `degParFiltering`, and `dedMultFiltering`, while input to class `Updating()` is: `spdUpdate`, `n_bits`, `bin_pt`, `degParUpdate`, `dedMultUpdate`, `sVec`, and `U`. When `MVDR()` is instantiated, `Filtering()`, `Updating()`, and the classes contained by them, are also instantiated with the corresponding parameters. Then, *PyGen* converts the instantiated Python objects into the corresponding designs in *System Generator*.

We consider the design requirement specified in [3], which demands a throughput of more than one million data samples per second and a data precision of more than 14 bit to ensure the performance of the beamforming algorithm. Under this requirement, we analyze the performance models associated with the multipliers and the adders and conclude that: due to the availability of dedicated multipliers, multiplication is more energy efficient than addition/subtraction under the design requirement in [3]. Thus, a four-multiplier architecture, instead of a three-multiplier one, is used for complex multiplication. Similarly, by comparing the computation costs and the storage costs, $\underline{U}^H \cdot \underline{U}$ is computed at compile time and is stored in the Block RAMs on the device. We do not provide parameters to change the architecture of these modules.

Besides, even though operating speeds of the tasks are parameterized, there are issues in simulating them. Since *filtering* and *coefficient updating* are operating at different speeds, their hardware implementations are placed into different clock domains on Xilinx FPGAs. This is implemented using DCM (Digital Clock Management) modules on the device. However, simulation of multiple clock domains is not supported by the current version of MATLAB/Simulink. Thus, we must use the same operating speed for both the filtering task and the coefficient updating task when performing arithmetic simulation in MATLAB/Simulink. We set them to their actual operating speeds before we translate the design into HDL implementations. The HDL implementations of these two modules are generated using *System Generator*. After that, the `MVDR()` class generates a top-level HDL code that combines the two implementations according to their actual operating speeds. The `MVDR()` class also combines the testbenches generated by *System Generator* and generates a top-level HDL testbench code for simulation.

3.3. Energy Estimation and Profiling

Referring to the design requirements in [3], we set `n_bits`=16, `bin_pt`=6. There are eight antenna elements in the system and thus `sVec`=8. Besides, the design constraints are: the incoming data rate is 100 million samples per second per antenna element; the execution rate of the *coefficient updating* task can be much slower than that of the *filtering* task; there are 5120 slices, 40 18x18-bit dedicated multipliers, and 40 BRAMs on Virtex-II xc2v1000 FPGA, our target device.

Since the filtering task needs to process each set of data samples and demands a large amount of computation power, we set `degParFiltering`=8. Thus, if it runs at 100MHz (`spdFiltering`=100), it can support the specified input data rate. Since increasing the degree of parallelism of a design greatly increases the required resources, by estimating the area requirement of the two tasks and comparing it with what is available on the target device, we set `degParUpdate`=1.

We provide the above settings and test data from Simulink simulation to the performance models associated with all the objects in the designs. By summing up the output from these models, we obtain coarse estimates of the performance of various designs, which are shown in Figure 2. The input to the beamforming application is streaming data. Also, our design is pipelined and all the modules are active throughout the processing. Therefore, *the energy efficiency of a design is measured as its average power consumption.*

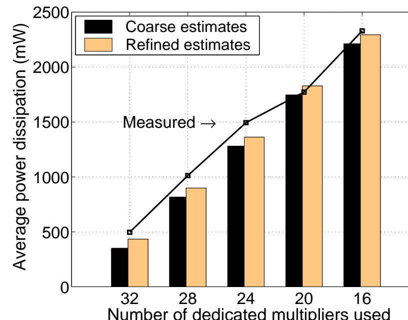

Fig. 2. Estimates of various designs for the filtering task

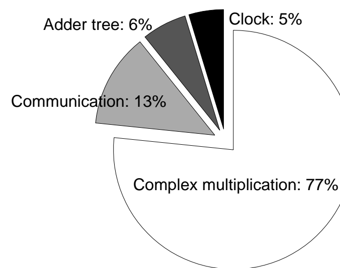

Fig. 3. Energy profile of the first design in Figure 2

Since the performance models only account for the energy dissipation of the FPGA modules represented by them, the coarse estimates do not capture the communication costs for sending data between the modules. For example, they do

not capture the energy dissipation for sending data from the I/O pads to the filtering task and that for sending data from the filtering task to the updating task. Using the coarse estimates, we identify several *good* candidate designs with low energy dissipation. For each of these designs, we perform low-level simulation, profile its energy dissipation, and use this information to refine the coarse energy estimates. Figure 3 shows that the communication costs mentioned above can account for 13% of the total energy dissipation. We use these profiling results to update the coarse estimates and obtain refined estimates. The average accuracy of our estimates is improved from 17.4% to 9.4% for the five designs shown in Figure 2 by comparing their estimates and the results from low-level simulation (the *Measured* data). We have performed similar estimates and analysis for the updating task. They are not included in this paper due to space limitation.

3.4. Optimization for Energy Efficiency

The *Case 1* in Figure 4 shows the performance of various designs when the filtering task runs at 100 MHz and the coefficient updating task runs at 50 MHz (`spdUpdate`=50). The labels for the X axis show the numbers of dedicated multipliers used by the designs. For example, "32+7" represents a design with `dedMultFiltering`=32 and `dedMultUpdate`=7. Due to the latency caused by the design of the coefficient updating task, this means that the coefficients of the filter are updated once every 16 samples. The *Case 2* in Figure 4 shows the performance of different designs when `spdFiltering`=`spdUpdate`=100, in which the coefficients are updated once every 8 samples.

Note that from the estimates, the most energy efficient designs for the two tasks are those with `dedMultFiltering`=32 and `dedMultUpdate`=11, respectively. However, these two designs cannot fit into our target device simultaneously. Thus, their combination is ignored in the optimization process. Also, *quiescent power* (power consumption of the device when there is no switching activity on it) and the output power from the I/O pads are not shown since they are fixed once the target device and the output data requirements are determined and cannot be optimized in our design flow.

We can see from our results that, under different application requirements, the most energy efficient designs are different. In Figure 4, the design labeled as "32+7" for *Case 1* and the design labeled as "28+11" for *Case 2* have the least amount of average power consumption, respectively. This is because as rate of updating the coefficients increases, more computation is performed by the updating task. We need to allocate more energy efficient resource such as dedicated multipliers to the updating tasks in order to improve energy efficiency. Considering the four candidate designs shown in Figure 4, up to 51% energy reduction can be achieved by doing so.

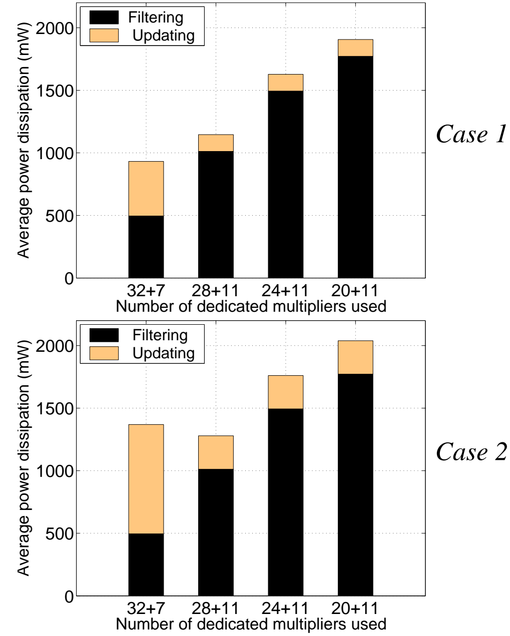

Fig. 4. Energy efficiency of various designs on Xilinx Virtex-II xc2v1000 FPGA (*Case 1*: filtering at 100 MHz while updating at 50 MHz; *Case 2*: both at 100 MHz)

4. CONCLUSION AND FUTURE WORK

Development of an energy efficient adaptive beamforming application on FPGAs using a MATLAB/Simulink based design flow is presented in this paper. Our technique can also be applied to the development of other adaptive beamforming applications, such as QR-based RLS (Recursive Least Squares) adaptive beamforming. They use more complex algorithms and offer more design trade-offs.

5. REFERENCES

[1] Altera, Inc., `http://www.altera.com`.

[2] S. Choi, J.-W. Jang, S. Mohanty, V. K. Prasanna, "Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures," *Engr. of Reconf. Systems/Algorithms (ERSA)*, 2002.

[3] M. Devlin, "How to Make Smart Antenna Arrays," *Xilinx XCell Journal*, Issue 45, 2003.

[4] C. Dick, "The Platform FPGA: Enabling the Software Radio," *Software Defined Radio Technical Conference and Product Exposition (SDR)*, 2002.

[5] S. Haykin, *Adapt. Filter Theory*, Prentice Hall, 2002.

[6] J. Ou, S. Choi, G. Govindu, and V. K. Prasanna, "Creating Parameterized and Energy-Efficient System Generator Designs," *Military & Aerospace Application of Prog. Logic Devices (MAPLD)*, 2003.

[7] Python, `http://www.python.org`.

[8] Xilinx, Inc., `http://www.xilinx.com`.