SPHERE DETECTOR FOR 802.16E BROADBAND WIRELESS SYSTEMS IMPLEMENTATION ON FPGAS USING HIGH-LEVEL SYNTHESIS TOOLS

Juanjo Noguera (Xilinx, Dublin, Ireland; juanjo.noguera@xilinx.com), Stephen Neuendorffer (Xilinx, San Jose, CA, USA; stephen.neuendorffer@xilinx.com), Sven Van Haastregt (Leiden University, Leiden, The Netherlands; svhaastr@liacs.nl), Jesus Barba (University of Castilla-La Mancha, Ciudad Real, Spain; jesus.barba@uclm.es), Kees Vissers (Xilinx, San Jose, CA, USA; kees.vissers@xilinx.com), Chris Dick (Xilinx, San Jose, CA, USA; chris.dick@xilinx.com)

ABSTRACT

In this paper we explain the implementation of a sphere detector for spatial multiplexing in broadband wireless systems using High-level Synthesis (HLS) tools. These modern FPGA design tools accept C/C++ descriptions as input specifications, and automatically generate a Register Transfer Level (RTL) description for FPGA implementation using traditional FPGA implementation tools.

We have used AutoESL's AutoPilot HLS tool to implement this demanding algorithm on a Virtex-5 running at a clock frequency of 225MHz. The obtained results show that these modern high-level synthesis tools produce Quality of Results (QoR) competitive to the ones obtained using a traditional RTL design approach, while significantly abstracting the designer from the low-level FPGA implementation details.

1. INTRODUCTION

Spatial division multiplexing MIMO processing significantly increases the spectral efficiency, and hence capacity, of a wireless communication system: it is a core component of next generation wireless systems, for example, WiMAX and other OFDM-based wireless communication standards. Spatial multiplexing MIMO processing is a computationally intensive application that implements highly demanding signal processing algorithms. A specific example of spatial multiplexing in MIMO systems is Sphere decoding (SD), which is a complexity-efficient method to solve the MIMO detection problem, while maintaining a bit-error rate (BER) performance comparable to the optimal maximum-likelihood (ML) detection algorithm. However, even this reduced-complexity algorithm is generally not feasible to implement on a DSP processor in real-time.

Field Programmable Gate Arrays (FPGAs) are an attractive target platform for the implementation of complex DSP-intensive algorithms, like the Sphere Decoder. Modern FPGAs are high-performance parallel computing platforms that provide the high-performance of dedicated hardware solutions, while keeping the flexibility of programmable DSP processors. There are several studies showing that FPGAs could achieve 100X higher performance and 30X better cost-performance than traditional DSP processors in several signal processing applications [4].

Despite this tremendous performance advantage, FPGAs are not generally used in wireless signal processing since they are perceived as devices difficult to use for traditional DSP programmers. The key barrier for the widespread adoption of FPGAs in wireless applications is the traditional hardware-centric design-flow and tools. Currently, the use of FPGAs requires significant hardware design experience, like for example, being familiar in using hardware description languages (e.g., VHDL, Verilog).

Recently, new High-level Synthesis Tools [3] have become available as design tools for FPGAs. These design tools take as input a high-level algorithm description and generate RTL that can be used with standard FPGA implementation tools (e.g., Xilinx ISE/EDK). These tools offer an increase in the design productivity and reduction of the development time, while producing good Quality of Results [2]. This paper describes the FPGA implementation of a complex wireless algorithm on a modern FPGA (i.e., sphere detector for spatial multiplexing MIMO in 802.16e systems) using High-level Synthesis Tools. We have used AutoESL's AutoPilot HLS tool to target a Xilinx Virtex-5 running at 225MHz. We present a comprehensive study reporting on our experiences in using HLS tools for this particular wireless algorithm and compare the results to an implementation generated using on a traditional FPGA hardware-centric design approach.

2. SPHERE DECODER

Sphere detection is a prominent method of simplifying the detection complexity in spatial multiplexing systems while maintaining BER performance comparable with optimum maximum-likelihood (ML) detection.

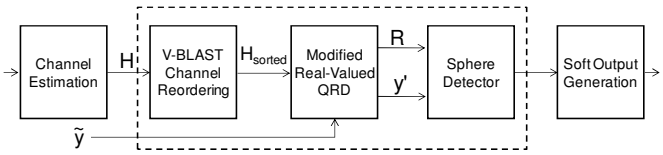

Figure 1: Block Diagram for Sphere Decoder

The block diagram of the MIMO 802.16e wireless receiver is shown in the Figure 1. It is assumed that the channel matrix is perfectly known to the receiver which can be accomplished by classical means of channel estimation. After channel reordering and QR decomposition, the sphere detector (SD) is applied. In preparation for engaging a soft-input-soft-output channel decoder (e.g. Turbo decoder), soft outputs are produced by computing the log-likelihood ratio (LLR) of the detected bits. A detailed explanation of this algorithm can be found in [1]. Following we briefly introduce the key three building blocks in this algorithm.

2.1. Channel Matrix Reordering

The order in which the antennas are processed by the sphere detector has a profound impact on the BER performance. So prior to sphere detection, channel reordering is applied. By utilizing a channel matrix pre-processor that realizes a type of successive interference cancellation similar in concept to that employed in BLAST (Bell Labs Layered Space Time) processing, the detector achieves close to ML performance. The method implemented by the channel reordering determines the optimum detection order of columns of the complex channel matrix over several iterations. Depending on the iteration count, the row with the maximum or minimum norm is selected. The row with the minimum Euclidian norm represents the influence of the strongest antenna while the row with the maximum Euclidian norm represents the influence of the weakest antenna. The novel approach first processes the weakest stream. All subsequent iterations process the streams from highest to lowest power.

To meet the high data rate requirements, the channel ordering block is realized using the pipelined architecture shown in Figure 2, which processes 5 channels in a Time Division Multiplexing (TDM) approach. This approach provided more processing time between the matrix elements of the same channel while sustaining high data throughput. The calculation of the G matrix is the most demanding component in Figure 2. The heart of the process is *Matrix Inversion* which is realized using QR decomposition (QRD). A common method for realizing QRD is based on Givens Rotations. The proposed implementation performs the complex rotations in the *Diagonal* and *OffDiagonal* cells, which are the fundamental computations units in the systolic array we are using.

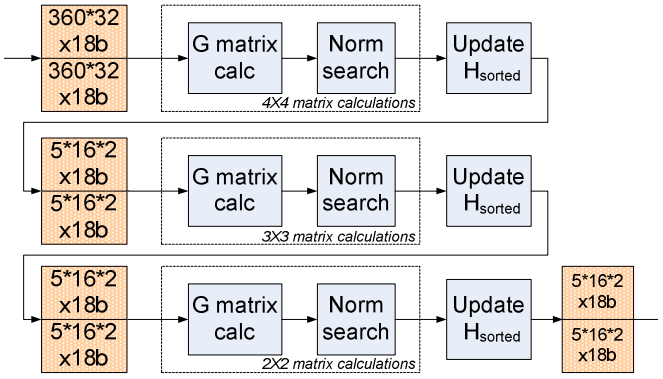

Figure 2: Iterative channel matrix reordering algorithm

2.2. Modified Real-Valued QR decomposition

After obtaining the optimal ordering of the channel matrix columns, the QR decomposition (QRD) on the real-valued matrix coefficients is applied. The functional unit used for this QRD processing is similar to the QRD engine designed to compute the inverse matrix, but with some modifications. The input data in this case are real valued and the systolic array structure has a correspondingly higher degree. In order to meet the desired timing constraints the input data consumption rate had to be 1 input sample per clock cycle. This introduced challenges around processing latency problems which couldn't be addressed with a 5-channel TDM structure. The number of channels in a TDM group was increased to 15 to provide more time between the successive elements of the same channel matrix.

2.3. Sphere Detector (SD)

The iterative sphere detection algorithm can be viewed as a tree traversal with each level of the tree *i* corresponding to processing symbols from the *ith* antenna. The tree traversal can be performed using several different methods. In our implementation we selected a breadth-first search due to the attractive hardware-friendly nature of the approach. At each level only the K nodes with the smallest partial Euclidian distance (*Ti*) are chosen for expansion. This type of detector is called a K-best detector.

The norm computation is done in the PED blocks of the sphere detector. Depending on the level of the tree, three different PED blocks are used: the root node PED block calculates all possible PEDs (tree level index is i =M =8). The second level PED block computes 8 possible PEDs for

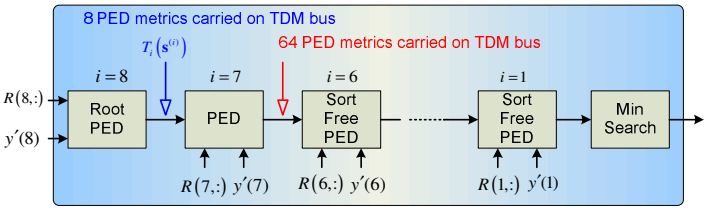

Figure 3: Sphere Detector Processing Pipeline

each of the 8 survivor paths generated in the previous level. This will give us 64 generated PEDs for the tree level index i = 7. The third type of PED block is used for all other tree levels which compute the closest-node PED for all PEDs computed on the previous level. This will fix the number of branches on each level to K = 64, thus propagating to the last level i =1 and producing 64 final PEDs along with their detected symbol sequences.

The pipeline architecture of the SD allows data processing on every clock cycle, thus the number of PED blocks necessary at every tree level is only one. The total number of PED units is equal to the number of tree levels, which for 4×4 64-QAM, is 8. The block diagram of the SD is illustrated in the Figure 3.

2.4. FPGA Performance Implementation Targets

The target FPGA device is a Xilinx Virtex-5 FPGA, with a target clock frequency of 225MHz. The channel matrix is estimated for every data subcarrier which limits the available processing time for every channel matrix. For the selected clock frequency and a communication bandwidth of 5MHz (corresponding to 360 data sub-carriers in a WiMAX system), the available number of processing clock cycles per channel matrix interval is calculated as follows:

$$num\_cycles = \frac{(102.9us/360)}{1/225MHz} \cong 64 \qquad (10)$$

As mentioned earlier, the most computationally demanding configuration with 4×4 antennas and 64-QAM modulation scheme has been designed. The achievable raw data rate in this case is 83.965Mbps.

3. HIGH-LEVEL SYNTHESIS FOR FPGA

High-level synthesis tools take as their input a high-level description of the specific algorithm to implement and generate the RTL description of FPGA implementation.

Modern high-level synthesis tools accept *untimed* C/C++ descriptions as input specifications. These tools give two interpretations to the same C/C++ code: (1) sequential semantics for input/output behavior; and (2) architecture specification based on C/C++ code and compiler directives. Based on the C/C++ code, compiler directives and target throughput requirements, these high-level synthesis tools generate high-performance pipelined architectures. Among other features, high-level synthesis tools enable automatic pipeline stages insertion, resource sharing to reduce FPGA resource utilization. In summary, high-level synthesis tools raise the level of abstraction for FPGA design, and make transparent the time-consuming and error-prone RTL design tasks. We have focused on using C++ descriptions, with the goal of leveraging C++ template classes to represent arbitrary precision integer types and template functions to represent parameterized blocks in the architecture.

The overall design approach is shown in Figure 4, where the starting point is a reference C/C++ code that could have been derived from a MATLAB functional description. As illustrated in this figure, the first step in implementing an application on any hardware target is often to restructure the reference C/C++ code. By "restructuring," we mean rewriting the initial C code (which is typically coded for clarity and ease of conceptual understanding rather than for optimized performance) into a format more suitable for the target processing engine. For example, on a DSP processor it may be required to rearrange an application's code so that the algorithm makes an efficient use of the cache memories. When targeting FPGAs, this code restructuring might involve, for example, rewriting the code so it represents an architecture specification that can achieve the desired throughput, or rewriting the code to make efficient use of the specific FPGA features like embedded DSP macros.

The *functional* verification of this implementation C/C++ code is achieved using traditional C/C++ compilers (e.g., gcc) and reusing C/C++ level testbenches developed for the verification of the reference C/C++ code. The implementation C/C++ code is the main input to the high-level synthesis tools. However, there are additional inputs to the high-level synthesis tools that significantly influence the generated hardware, its performance and number of FPGA resources used. Two essential *constraints* are the target FPGA family (i.e., technology) and target clock frequency, which obviously have an effect on the number of pipeline stages in the generated architecture. Additionally, high-level

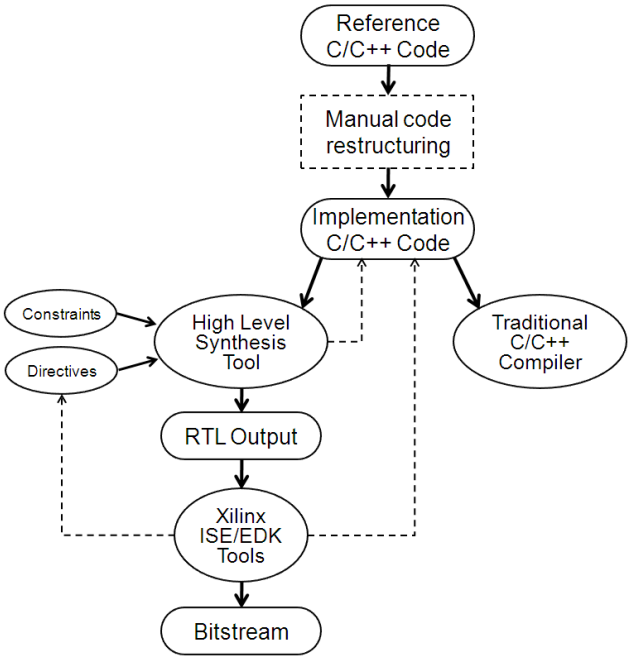

Figure 4: High-level Synthesis for FPGAs

synthesis tools accept *compiler directives* (e.g., *pragmas* inserted in the C/C++ code). There are different types of directives, which can be applied to different sections of the C/C++ code. For example, there are directives that are applied to loops (e.g., loop unrolling), while other directives can be applied to arrays (e.g., to specify which FPGA resource must be used to the implementation of the array).

Based on all these inputs, the high-level synthesis tools generate an output architecture (RTL) and report the throughput of the generated architecture. Depending on this throughput, then the designer can modify the directives and/or the implementation C/C++ code. If the generated architecture meets the required throughput, then the output RTL is used as the input to the FPGA implementation tools (ISE/EDK). The final achievable clock frequency and number of FPGA resources used is reported only after running logic synthesis and place&route. If the design does not meet timing or the FPGA resources are not the expected ones, the designer should modify the implementation C/C++ code or the compiler directives.

4. HIGH-LEVEL SYNTHESIS IMPLEMENTATION OF SPHERE DECODER

We have implemented the key three building blocks of the WiMAX sphere decoder shown in Figure 1 using AutoPilot 2010.07.ft from AutoESL. It is important to emphasize that the algorithm is exactly the algorithm described in [1], and hence has exactly the same BER. In this section we give specific examples of code re-writing and compiler directives that we have used for this particular implementation.

4.1. Design approach: Iterative C/C++ refinement

The original reference C code, derived from a MATLAB functional description, included 2000 lines of code (aprox.), including synthesizable and verification C code. It contains only fixed-point arithmetic using C built-in data types. All the required floating point operations (e.g., *sqrt*) have been approximated by a FPGA-friendly implementation.

In addition to the reference C code describing the functions to synthesize in the FPGA, there is a complete C-level verification testbench. The input test vectors, as well as the golden output reference files, were generated from the MATLAB description. The original C reference code is bit-accurate with the MATLAB specification, and passes the entire regression suite consisting of multiple data sets.

This reference C code has gone through different types of code restructuring. As examples, Figure 5 shows three instances of code restructuring that we have used, which are explained in the following subsections. A key concept to keep in mind is that the C-level verification infrastructure has been re-used to verify any change to the implementation C/C++ code. All verification has been carried out at the C-level, not at the RTL level, avoiding time-consuming RTL simulations and hence, contributing to the reduction in the overall development time.

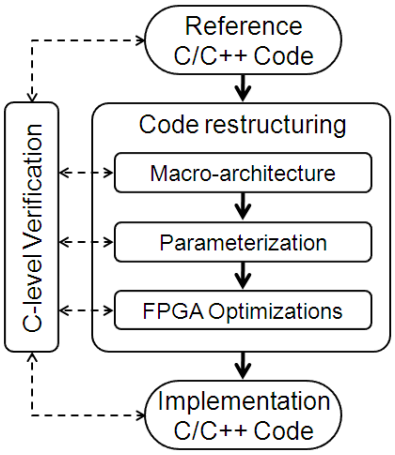

Figure 5: Iterative C/C++ refinement design approach

4.2. Macro-architecture specification

Probably the most important code refactoring example is to rewrite the C/C++ code to describe the *macro-architecture* that efficiently would implement a specific functionality. In other words, the designer is accountable for the *macro-architecture* specification, while the high-level synthesis tools are in charge of the *micro-architecture* generation. This type of code restructuring has a major impact on the obtained throughput and quality of results.

In the case of the sphere decoder, there are several instances of this type of code restructuring. For example, to meet the high throughput of the channel ordering block, the designer should describe in C/C++ the macro-architecture shown in Figure 2. Such C/C++ code would consist of several function calls communicating using arrays. The high-level synthesis tools might automatically translate these arrays in ping-pong buffers to allow parallel execution of the several matrix calculation blocks in the pipeline. Another example of code restructuring at this level would be to decide how many number of channels should be employed in the TDM structure of a specific block (e.g., 5 channels in the Channel Matrix Reordering block, or 15 channels in the Modified Real-Valued QR decomposition block.

Figure 6 is a specific example of macro-architecture specification. This snipped of C++ code describes the sphere detector block diagram shown in Figure 3. We can observe a pipeline of nine function calls, each one representing a block as shown in Figure 3. The communication between functions is achieved through arrays, which are mapped to streaming interfaces (not embedded BRAM memories in the FPGA) by using the appropriate directives (*pragmas*) in lines 5 and 7.

```
1: void sphere_detector_top (…) {
2:   #pragma AP DATAFLOW
3:   // PED streams between pipeline blocks
4:   ap_int<18>    PED_7[RVD_MODULATION];
5:   #pragma AP ARRAY_STREAM variable=PED_7 depth=1 stream
6:   ap_int<4>     symb_7[RVD_MODULATION];
7:   #pragma AP ARRAY_STREAM variable=symb_7 depth=1 stream
8:   main_label:{
9:     RootPED(r_7, y_7, ..., symb_7);
10:    PED(r_6, y_6, ..., symb_7, symb_6);
11:    SortFreePED<5,2>(r_5, y_5, ..., symb_6, symb_5);
12:    SortFreePED<4,3>(r_4, y_4, ..., symb_5, symb_4);
13:    SortFreePED<3,4>(r_3, y_3, ..., symb_4, symb_3);
14:    SortFreePED<2,5>(r_2, y_2, ..., symb_3, symb_2);
15:    SortFreePED<1,6>(r_1, y_1, ..., symb_2, symb_1);
16:    SortFreePED<0,7>(r_0, y_0, ..., symb_1, symb_0);
17:    // find minimum PED
18:    min_finder(PED_0, symb_0, min_PED, min_symb_list);
19:  }
20:}
```

Figure 6: Sphere Detector macro-architecture description

4.3. Parameterization

Parameterization is another key example of code re-writing. We have extensively leveraged C++ template functions to represent parameterized modules in the architecture.

In the implementation of the sphere decoder, there are several examples of this type of code rewriting. A specific example would be the different matrix operations used in the channel reordering block. The *Matrix Calculations* blocks (4x4, 3x3 and 2x2) showed in Figure 2 use different types of matrix operations like *Matrix Inverse* or *Matrix Multiply*. These blocks are coded as C++ template functions with the dimensions of the matrix as template parameters.

Figure 7 shows the C++ template function for *Matrix Multiply*. In addition to the matrix dimension, this template function has a third parameter *MM_II* (*Initiation Interval* for Matrix Multiply), which is used to specify the number of clock cycles between two consecutive loop iterations. See directive (*pragma*) in line 9, which is used to parameterize the required throughput for a specific instance. This is a

```
1:  template<int X_DIMENSION, int Y_DIMENSION, int MM_II>
2:  void matrix_multiply_(...) {
3:  #pragma AP ARRAY_PARTITION variable=chunk_in_re dim=1
4:  #pragma AP ARRAY_PARTITION variable=chunk_in_im dim=1
5:   // matrix multiplication of a A'*A matrix
6:   for (index_a = 0; index_a < TDM_CHUNKS; index_a++) {
7:     for (index_b = 0; index_b < X_DIMENSION; index_b++) {
8:       for (index_c = 0; index_c < Y_DIMENSION; index_c++) {
9:  #pragma AP PIPELINE II = MM_II
10:              <loop body>
11:   } } }
12: }
```

Figure 7: Example of code parameterization

```
1:  template <int Wa, int Wb, int Wc>
2:  ap_int<36> SUBMUL(ap_int<Wa> a, ap_int<Wb> b, ap_int<Wc> c) {
3:    #pragma AP LATENCY max=2
4:    #pragma AP INTERFACE ap_none port=return register
5:
6:    ap_int<36> c_36 = c; // sign extension
7:    return c_36-a*b;
8:  }
```

Figure 8: FPGA optimization for DSP48 utilization

really important feature, since it has a major impact on the generated micro-architecture. That is, the ability of the high-level synthesis tools to exploit resource sharing, and hence, reducing the FPGA resources used in the implementation. For example, just by modifying this *Initiation Interval* parameter and using exactly the same C++ code, the high-level synthesis tools automatically achieve different levels of resource sharing in the implementation of the different *Matrix Inverse* (4x4, 3x3, 2x2) blocks.

4.4. FPGA Optimizations

FPGA optimization is the last example of code rewriting. The designer can rewrite the C/C++ code to more efficiently utilize specific FPGA resources, and hence, improve timing and reduce area. There are two very specific examples of this type of optimizations: (1) bit-widths optimizations; and (2) efficient use of embedded DSP blocks (i.e., DSP48s). The reference C/C++ code was written using built-in C/C++ data types (e.g., short, int), while the design uses 18-bit fixed point data types to represent the matrix elements. We have leveraged C++ template classes to represent arbitrary precision fixed-point data types, hence reducing FPGA resources and minimizing impact on timing.

Figure 8 is a C++ template function that implements a multiplication followed by a subtraction, where the width of the input operands is parameterized. These two arithmetic operations can be mapped into a single embedded DSP block (i.e., DSP48 block). By efficiently using DSP48s, timing is improved and FPGA resource utilization is minimized. In Figure 8, we can also observe two directives that instruct the high-level synthesis tool to use a maximum of two cycles to schedule these operations and use a register for the output return value.

5. PRODUCTIVITY METRICS

5.1. Development Time

In Figure 9 we plot how the size of the design (i.e., FPGA resources) generated using AutoESL's AutoPilot evolves over time and compare it to a traditional SystemGenerator (i.e., RTL) implementation. It is important to observe in this figure that by using high-level synthesis tools we are able to implement *many* valid solutions, which differ in size over time. On the other hand, there is *only one* RTL solution,

which requires of a long development time. Many solutions can be obtained using high-level synthesis tools. Depending on the amount of code restructuring, the designer can trade-off how fast to get a solution versus the size of that solution.

We have observed that it is relatively quick to obtain several *Fast* solutions, which use significant more FPGA resources (i.e., area) than the traditional RTL solution. On the other hand, the designer might decide to generate many more *Expert* solutions by implementing more advanced C/C++ code restructuring techniques (e.g., FPGA-specific optimizations) to reduce FPGA resource utilization.

Finally, and since all verification has been carried out at the C/C++ level, not at the RTL-level, we avoided the time-consuming RTL simulations. We found that doing the design verification at the C/C++ level significantly contributed to the reduction in the overall development time.

5.2. Quality of Results

In Figure 10, we compare final FPGA resource utilization and overall development time for the complete sphere decoder implemented using high-level synthesis tools and the reference System Generator implementation, which is basically a structural RTL design, explicitly instantiating FPGA primitives, like for example, DSP48 blocks. Please, note that the development time for AutoESL includes learning the tool, producing results, design space exploration and detailed verification.

To have accurate comparisons, we have re-implemented the reference RTL design using the latest Xilinx ISE 12.1 tools targeting a Virtex-5 FPGA. The RTL generated by AutoESL's AutoPilot has also been implemented using ISE 12.1 targeting the same FPGA. From table in Figure 10, we can observe that AutoESL's AutoPilot achieve significant savings in FPGA resources, which is mainly achieved through resource sharing in the implementation of the matrix inverse blocks. We can also observe a significant reduction

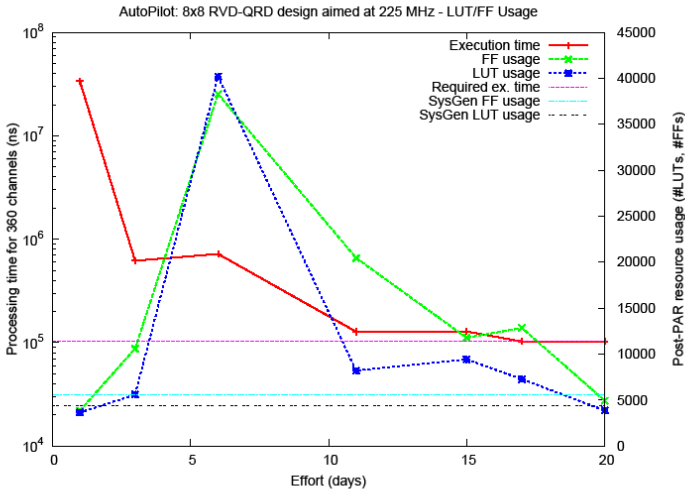

Figure 9: Development time

Metric	SysGen Expert	AutoESL Expert	% Diff
Development Time (weeks)	16.5	15	-9%
LUTs	27,870	29,060	+4%
Registers	42,035	31,000	-26%
DSP48s	237	201	-15%
18K BRAM	138	99	-28%

Figure 10: Quality of results

in the number of registers and a slightly higher utilization of Look-up Tables (LUTs). This result is partially due to the fact that delay lines are mapped onto SRL16s (i.e., LUTs) in the AutoESL implementation, while the delay lines are implemented using registers in the SystemGenerator implementation. There are other modules where we traded-off BRAMs for LUTRAM, resulting in lower BRAM usage in the channel preprocessor.

6. CONCLUSIONS

In this paper we have presented the implementation of a complex and demanding wireless MIMO receiver using a high-level synthesis tool targeting a Xilinx FPGA.

This evaluation has demonstrated that AutoESL's AutoPilot achieves significant abstractions from low-level FPGA implementation details (e.g., timing and pipeline design), while producing Quality of Results (QoR) highly competitive to the ones obtained using a traditional RTL design approach. C/C++ level verification contributes to the reduction in the overall development time by avoiding time-consuming RTL simulations However, obtaining excellent results for complex and challenging designs requires good macro-architecture definition and FPGA tools knowledge (e.g., understand and interpret FPGA tool reports).

7. REFERENCES

[1] Chris Dick *et al.*, "FPGA Implementation of a Near-ML Sphere Detector for 802.16E Broadband Wireless Systems" SDR-Conference'09, Dec. 2009.

[2] Berkeley Design Technology, Inc., "High-Level Synthesis Tools for Xilinx FPGAs", White paper 2010. [Online]: http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf

[3] Grant Martin, Gary Smith. "High-Level Synthesis: Past, Present, and Future," IEEE Design and Test of Computers, July/August 2009.

[4] Berkeley Design Technology, Inc., "FPGAs for DSP", White paper 2007.

[5] K. Denolf, S. Neuendorffer, K. Vissers, "Using C-to-gates to program streaming image processing kernels efficiently on FPGAs"; FPL conference, Sep. 2009.