High-Level Modeling and Synthesis for Embedded FPGAs

Xiaolin Chen†, Shuai Li†, Jochen Schleifer‡, Thomas Coenen‡,
Anupam Chattopadhyay†, Gerd Ascheid†, Tobias G. Noll‡
† Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
‡ Chair of Electronic Engineering and Computer Systems, RWTH Aachen University, Germany
Email: chenxi@ice.rwth-aachen.de, {schleifer, coenen, tgn}@eecs.rwth-aachen.de

Abstract—The fast evolving applications in modern digital signal processing have an increasing demand for components which have high computational power and energy efficiency without compromising the flexibility. Embedded FPGA, which is the customized FPGA with heterogeneous fine-grained application specific operations and routing resources, has shown significantly improved efficiency in terms of throughput, power dissipation and chip area for the target application domain. On the other hand, the complexity of such architecture makes it difficult to perform an efficient architecture exploration and application synthesis without tool support. In this work, we propose a framework for the design of embedded FPGA (eFPGA) architectures, which is extended from an existing framework for Coarse-Grained Reconfigurable Architectures (CGRAs). The framework is composed of a high-level modeling formalism for eFPGAs to explore the mapping space, and a retargetable application synthesis flow. To enable fast design space exploration, a force-directed placement algorithm is proposed. Finally, we demonstrate the efficacy of this framework with demanding application kernels.

I. INTRODUCTION

Modern System-on-Chips (SoCs) for the embedded application domain contain a wide variety of components to meet the tight constraints of performance and energy dissipation without compromising the flexibility. Different types of architectural realizations are being investigated along with the traditional programmable processors and ASICs. An interesting development in this field is to have reconfigurable architectures e.g. FPGA or CGRA, as processing components on Multiple Processor System-on-Chips (MPSoCs), or to combine with application-specific processors [7][8][9]. An FPGA is usually constructed from look-up table (LUT) based logic elements (LEs) and general purpose routing resources. This gives the full flexibility to map any application, but is poor in performance and efficiency. On the other hand, the word-level computation units and routing resources of CGRA can provide high performance and efficiency while sacrificing the flexibility for the target applications. As a compromise, we may tune FPGA to the requirements of a specific application domain, or compose CGRA of finer grained application-specific operators and routing resources. Such architecture can be termed as application-specific/embedded FPGA. Significantly improved efficiency of such architecture compared to FPGAs has been shown in [1][3] for arithmetic kernels.

For the design of CGRAs or FPGAs, tool support[5][6] is usually available. For eFPGA architectures, design tools are still lacking. For this reason, most of the initial work on eFPGA is usually based on optimized FPGA macros[8][10][11] targeting a specific application domain. For these designs, because the architecture is already fixed, efficient synthesis tools can be designed for mapping applications onto the FPGA macro. However, when switching to a different application domain, re-design of the architecture and the tools is needed.

Recently, some work have emerged to take a different approach, in which a parameterizable template model is used as the starting point for designing eFPGA architectures[2][9].

In [2], an eFPGA architecture template is proposed targeting the arithmetic-oriented applications. The template is based on a RTL implementation of the proposed eFPGA architecture with a number of architecture parameters which can be tuned by the designer. Typical parameters include e.g. the number of logic elements (LEs) in a cluster, functionality and local connections of LEs, the number of routing tracks and configuration of the routing switch etc. Based on this template, a design flow down to physical implementation is available. By configuring the architecture parameters, design space exploration can be done to design proper architectures for different application domains. For mapping applications onto eFPGA, there is an attempt[4] to use a standard VLSI routing tool to perform the routing of nets between LEs. However, mapping and placement of LEs are done manually.

A commercial eFPGA architecture, which can also be seen as a template-based architecture, can be found in [9]. The soft core in synthesizable HDL is available to designers to make modifications according to their needs, and can be integrated as an IP in SoC. The eFPGA can be customized for target applications by adding highly optimized macros into the architecture template. A software tool suite for mapping applications onto the eFPGA as well as for hardware implementation is available.

The template-based modeling of eFPGA architecture has the following two limitations:
1. Limited design space by the parameterized model
2. Difficulty to add new architecture parameters
Although the architecture can be configured to fit different applications by tuning the architecture parameters, the design space is still limited by the availability of the parameters. The architecture parameters cannot always cover all the design points of such architecture. Moreover, for a given application domain, a lot of suitable parameter settings need to be found and fed to the associated CAD tools for application synthesis. Due to the strong coupling of the CAD tools with the architecture template, presence of any new parameters in the architecture may require significant modifications of the architecture template and the associated tool flow. These problems can only be solved by taking the high-level modeling approach.

**Contribution**

Modeling eFPGAs using a high-level language is based on two concerns.

1. Full flexibility to cover all architecture design points
2. Retargetable synthesis tool based on the high-level modeling language

Implementation of such eFPGA architecture usually calls for greedy physical optimization for the target applications. Considering the complexity of the architecture and the wide variety of design choices, only high-level modeling can provide designers the freedom to adjust architectural features to find the most proper architecture for the target applications, and thereby avoid restrictions of the parameterized design approach. Given an eFPGA specification modeled in high-level, an architecture representation can be figured out, which is composed of the defined LEs connected by routing resources. The application synthesis tools can be built in a generic manner based on the representation, so that the tools are retargetable to different architecture models.

The contribution of this work can be summarized as follows.

1. A high-level modeling language is proposed for eFPGA modeling.
2. A retargetable application synthesis flow is built up based on the high-level modeling language.
3. A force-directed placement and routing algorithm is proposed for application synthesis.

The rest of the paper is organized as follows. An overview of the eFPGA high-level modeling and synthesis framework is given in section II. In section III, the high-level modeling language is briefly introduced with some of the extended language features for the modeling of eFPGA architecture. In section IV, the retargetable application synthesis flow is described, and the proposed force-directed placement algorithm is focused. Section V reports the experiment results of the selected cases. The proposed force-directed algorithm is compared to both manually optimized results and the state-of-the-art simulated-annealing algorithm. The paper is concluded and future directions are mentioned in section VI.

**II. eFPGA Design Framework**

The eFPGA design framework (Fig. 1) can be divided into two parts, i.e. the implementation flow and the application synthesis flow. For implementation, the high-level eFPGA model is parsed to generate an intermediate representation of the eFPGA architecture. The architecture representation is subjected to an HDL back-end to generate the HDL model of the eFPGA. For application synthesis onto eFPGA, a bit-wise signal flow graph is mapped into LEs by the mapping tool. The netlist of LEs are placed and routed on the eFPGA using the architecture information from the implementation flow. As another input to the placement and routing, usage of I/O pads for the input and output signals needs to be specified. RTL simulation can be performed using the generated configuration bitstream from the placement and routing result for the generated eFPGA HDL model.

The LEs of the target eFPGA architecture[1] used in the case studies are optimized macros for arithmetic operations. In the current tool flow, a pattern library, which represents a set of optimized data-paths of the LEs, is used for mapping the signal flow graph to LE netlist. The signal flow graphs of the case studies is currently manually prepared in a format which can be read by the mapping tool. In the future, this can be generated directly from the application C code. The mapping algorithm is similar to the one used in [5]. For a general usage, the data path information of LEs needs to be parsed from the high-level eFPGA model and used by the mapping tool. The mapping tool needs to support a generic mapping for a data path of LE with LUTs and logic operations. These issues are planned as possible extensions in the future.

**III. Modeling of eFPGA**

The high-level modeling language of eFPGA is obtained by extending a high-level modeling formalism for CGRAs[5]. Using the high-level language, an eFPGA model is organized in three interrelated parts, which are ELEMENT, TOPOLOGY, and CONNECTIVITY. In this section, the eFPGA architecture proposed in [1] is used as the target architecture to show how it can be modeled using the high-level language.

**A. Logic Element**

A logic element (LE, Fig. 2), described using the keyword ELEMENT, is characterized by its functionality and interface. The functionality is described in the BEHAVIOR section inside an ELEMENT, whereas the interface is described through the PORT declaration. The input and output ports of a LE can
Resources can be modeled into a cluster (CB) and routing switch (RS). These routing resources within a cluster (e.g. clusters in Fig. 3) can receive different broadcast inputs using specific basic rules. As the example of Fig. 3, LE1 and LE2 can receive different broadcast inputs using specific basic rules. Besides the local connections and broadcast signals, routing resources within a cluster (e.g. clusters in Fig. 3) also include connection box (CB) and routing switch (RS). These routing resources can be modeled into clusters, in which specific routing can be defined in their connectivity section.

IV. APPLICATION SYNTHESIS ONTO eFPGA

The two stages for synthesizing an application onto eFPGA are the mapping stage and the placement and routing stage. As described in section II, a pattern library based mapping is currently used to map the signal flow graph to a netlist of LEs. The netlist is then passed to the placement and routing tools to get a minimum delay, non-congestion placement and routing on the eFPGA.
a placement and routing of the netlist on eFPGA, the force applied on a net between two LEs is calculated based on the Hooke’s law (Equation 1).

\[ F = -k \cdot l \]  

For \( l \), routing cost of the net can be used. The spring constant \( k \) is the other parameter to decide on the force of the net. In this work, we apply the slack ratio of the net [12] as the spring constant. The slack ratio is defined as the ratio between the delay of the longest path containing the current net and the delay of the critical path. It represents the criticality of the net. A net on the critical path always has the largest slack ratio, and therefore, a larger force tends to be applied on the net to shorten it more than the nets on non-critical paths.

Based on the force model, the forces applied on a LE from other LEs are analyzed, and the total forces on the \( x \) and \( y \) axes (e.g. \( F_{1,x} \) and \( F_{1,y} \) in Fig. 4) are calculated. The total forces on the \( n \)th LE on \( x \) and \( y \) axes are used to calculate the Equilibrium Position for the LE following Equation 2.

\[
\begin{align*}
\Delta x_n &= F_{n,x} \cdot C_{\text{scale}} \\
\Delta y_n &= F_{n,y} \cdot C_{\text{scale}} \\
x_{n,\text{equilibrium}} &= x_{n,\text{current}} + \Delta x_n \\
y_{n,\text{equilibrium}} &= y_{n,\text{current}} + \Delta y_n
\end{align*}
\]

The equilibrium position represents a location where the LE is more stable under the current force field applied by other LEs, and therefore the direction which the LE should be moved to. The relative coordinates of the equilibrium position to the current position is calculated by scaling the total forces equally on the \( x \) and \( y \) axes. This scaling factor is used to control the speed of movement of the LE on eFPGA.

C. Force-Directed Placement and Routing

Before placement and routing, the usage of the I/O pads for the input and output signals of the netlist is specified. This is because the I/O pads also apply forces to the LEs connected to them. It is used as a constraint for the placement and routing.

Based on the force model, the force-directed placement and routing algorithm is described in Algorithm 1. The algorithm starts from a random placement and routing of the netlist. Similar to the simulated-annealing algorithm in [13], a temperature parameter \( T \) is kept as the condition to terminate the algorithm. Within the iterations of temperature, total forces of each LE in the netlist are calculated based on the current placement and routing. The LEs are reorganized in the order of decreasing force. Within the inner loop, the LEs are visited following this order, which means the most unstable LE is handled first. The LE is moved to its equilibrium position, and nets related to this movement are rerouted. The movement is randomly accepted depending on the cost before and after the movement, and the current temperature. Similar to [13], the cost is calculated from the total delay of all nets and the routing congestion. Forces of LEs are recalculated if the movement is accepted. After all the LEs in the netlist are visited once in order in the inner loop, PathFinder[12] is used to solve congestions and temperature is updated. The LEs are only ordered once for each temperature. This is to make sure all LEs have the chance to be moved within the temperature iteration, which avoids the algorithm to be stuck in always handling bad movements of certain LEs.

![Fig. 4: Example of Force Model](image)

**Algorithm 1** Force-Directed Placement and Routing

1. \( G(V,E) \): Target architecture with placed I/O pads, \( N_{\text{nets}} \): Netlist of logic elements
2. \( \text{begin} \)
3. \( \text{CreateRandomPlacement} \left( N_{\text{nets}}, G(V,E) \right); \)
4. \( \text{R}=\text{GetRoutes} \left( N_{\text{nets}}, G(V,E) \right); \)
5. \( \text{C}=\text{CalcCost} \left( R, G(V,E) \right); \)
6. \( T=\text{CalcInitTemperature} \left( G(V,E), C \right); \)
7. \( \text{while } T > T_{\text{terminal}} \text{ do} \)
8. \( \text{F}=\text{CalcForceForLEs} \left( N_{\text{nets}}, G(V,E), R \right); \)
9. \( \text{N}_{\text{new}}=\text{OrderInDecreaseForce} \left( N_{\text{nets}}, F \right); \)
10. \( \text{LE}=\text{Nets}_{\text{new}}. \text{first}(); \)
11. \( \text{while } \text{N}_{\text{Move}} < \text{N}_{\text{LE}} \text{ do} \)
12. \( \text{R}_{\text{swap}}=\text{RipUpRelatedNets} \left( \text{LE} \right); \)
13. \( \text{eq}_{\text{nets}}=\text{CalcEquilibPos} \left( \text{LE}, G(V,E) \right); \)
14. \( \text{swap}=\text{CalcCost} \left( \text{R}_{\text{swap}}, G(V,E) \right); \)
15. \( \text{if } \text{AcceptMove} \left( \text{C}, \text{swap}, T \right) \text{ then} \)
16. \( \text{C}=\text{CalcCost} \left( \text{R}_{\text{swap}}, G(V,E) \right); \)
17. \( \text{else} \)
18. \( \text{RecalculateRoutesBeforeSwap} \left( G(V,E) \right); \)
19. \( \text{end if} \)
20. \( \text{end while} \)
21. \( \text{end} \)
22. \( \text{end while} \)
23. \( \text{end} \)
24. \( \text{PathFinder} \left( G(V,E) \right); \)
25. \( T=\text{UpdateTemperature}(); \)
26. \( \text{end} \)
27. \( \text{end} \)

V. Results

The proposed eFPGA design framework is evaluated by mapping three arithmetic oriented kernels onto the target eFPGA architectures[1]. These kernels are \( 5 \times 5 \) unsigned multiplier, FFT butterfly, and LDPC (Low-Density Parity-Check) check node. The multiplier case has a very regular data path. Through this simple case, some hints on how the proposed force-directed placement works are first shown. FFT butterfly and LDPC check node are two cases which represents two completely different types of application kernels. The FFT butterfly is a regularly organized pure data path kernel, widely used in digital signal processing. It is interesting to map it onto eFPGA as an accelerator in signal processing systems. In this kernel, four complex inputs, each having 5-bit real and image parts, are first accumulated. A complex multiplication is performed on the sum of the inputs with a twiddle factor to generate the complex output. The LDPC check node is a kernel combining data path with control path. This case is interesting because of the irregularity of its netlist, to test the limits of the placement and routing algorithms. The kernel only looks at the unsigned processing, which has 16 4-bit inputs and 16 4-bit outputs. It first performs a minimum search on the 16
inputs to get the first and second minimum values. The control bits from the minimum search, which reflect the comparison results, are used to generate further control signals through a multiplex tree. These control signals are then used to select between the two minimum values as the 16 outputs. These two cases are used to test how the proposed placement algorithm works for different types of application kernels. The netlist complexity of the cases in terms of number of used LEs and nets to be routed after mapping is shown in TABLE I.

A. eFPGA Modeling

The eFPGA macros used for the three cases are first modeled using the high-level language. The main parameters of the eFPGA configurations are listed in TABLE I. The LE architecture is modeled based on the LE in [1], and used for all three cases. The routing resources inside a cluster are modeled, which include two connection boxes with tracks and a routing switch. The eFPGAs are constructed using the modeled clusters. Global broadcast signals and local connection style between LEs are modeled specifically for each eFPGA configuration. The number of broadcast signals, tracks, LEs in a cluster, and clusters in eFPGA are selected based on the existing manual results of the cases. Note that, the usage of LEs on the eFPGA macros (i.e. ratio of LEs in the netlist to LEs on the eFPGA) for the FFT and LDPC cases are quite high, which are around 80% as listed in TABLE I.

B. eFPGA Synthesis

Without using tools, it usually takes a few days to generate the manual result even for a simple case like FFT butterfly. Based on the high-level eFPGA models, the existing manual results are regenerated using the manual mode of the synthesis tool, which are shown in the columns of “Manual” in TABLE II. The delay of nets are calculated based on the delay model defined in the previous section.

1) 5 × 5 Unsigned Multiplier: The layout of LEs using different placement modes are depicted in Fig. 5. The two 5-bit inputs are fed into the eFPGA from top and right of the eFPGA using broadcast signals (red line from input pads to LEs) and tracks (black lines from input pads to tracks on eFPGA border). For manual placement, because the LEs are placed exactly on the position to use the broadcast inputs, the inputs through tracks are not used. The brown blocks are the gated full adder array, and the blue blocks represent LEs of logic AND operation before the first row and column of the gated full adders. For the force-directed and simulated-annealing placement, the LEs cannot always be placed on the eFPGA border. For manual placement, because the LEs are fully used. The comparison of the layout of LEs between the manual and force-directed placement is shown in Fig. 6. In the manual placement, four complex inputs are fed from top into the eFPGA through broadcast signals and tracks. The inputs are first accumulated using Multi-Operand Adders (MOA), and multiplied with the complex twiddle factor. The complex twiddle factor is fed from the right using broadcast signals and tracks, and leaves one cluster row on the top for the MOA. The four multipliers are organized regularly so that the twiddle factor broadcasts are used. Multiplication results are added to get the outputs. The layout of LEs from force-directed is not as regular as the manual result. But as expected, the LEs belonging to the same block in FFT (e.g. multipliers or adders) are pulled close to each other to shorten the routing path. The multipliers and adders are positioned in a similar range of the eFPGA as the manual result. This is also partly because of the way to feed input signals. The MOAs are pulled to the first cluster row by the inputs from the top. The multipliers are affected.

2) FFT Butterfly: For FFT case, all 36 clusters on the eFPGA are fully used. The comparison of the layout of LEs between the manual and force-directed placement is shown in Fig. 5. In the manual placement, four complex inputs are fed from top into the eFPGA through broadcast signals and tracks. The inputs are first accumulated using Multi-Operand Adders (MOA), and multiplied with the complex twiddle factor. The complex twiddle factor is fed from the right using broadcast signals and tracks, and leaves one cluster row on the top for the MOA. The four multipliers are organized regularly so that the twiddle factor broadcasts are used. Multiplication results are added to get the outputs.
by the force from the twiddle factor, and positioned in the corresponding range where the twiddle factor is fed.

From the placement and routing results in TABLE II, force-directed is 27.5\% and 24.4\% worse in total net delay and critical path delay than the manual result, and 31\% and 51\% better than simulated-annealing. The run-time is another advantage of force-directed to simulated-annealing. For this case, force-directed runs for 20 minutes to get a congestion-free placement and routing. While for simulated-annealing, 12 congestions still exist after running for 1 day. An explanation for the congestions is the high usage ratio (80\%) of LEs on the eFPGA. Because there are not sufficient empty LEs for the random swapping in simulated-annealing, the movements of LEs are mostly rejected when the temperature gets low.

3) LDPC Check Node: Due to the limited space, the layout of LEs is not shown for this case. Because of the irregularity of the netlist, it is difficult to figure out the delay-optimal placement and routing. Therefore, the manual result is generated from an area-optimal placement, in which 509 LEs of the netlist use 104 clusters (520 LEs). In the force-directed placement, because the total delay of all nets is used as the optimization metric in the cost function, the algorithm will optimize the total net delay for a non-congestion placement and routing. This is why the total net delay of force-directed is close to the manual result in TABLE II. However, the critical path delay of the manual result is still better than force-directed. This is because critical path delay is not the optimization metric for force-directed, and the close positioning between clusters in the area-optimal manual placement can still produce a quite good critical path.

Compared to simulated-annealing, force-directed produces much better results in both total net delay and critical path (TABLE II). Because of the slow convergence speed of simulated-annealing, it requires very long time to achieve a reasonable result on this tightly used eFPGA. We keep it running for 2 days, but still cannot achieve a congestion-free result.

VI. CONCLUSION

In this paper, a high-level modeling and synthesis framework for embedded FPGAs is proposed. The framework is constructed by a high-level modeling and implementation flow with an application synthesis flow extended from a CGRA design framework. Different placement modes are supported for application synthesis, in which a force-directed placement and routing algorithm is proposed and shows improved results compared to the state-of-the-art simulated-annealing algorithm.

In future, we plan to integrate this framework with a physical implementation flow of eFPGA (e.g. [2]). This will enable the accurate and comprehensive study of eFPGA compared to other computing architectures.

REFERENCES