

Automatically mapping applications to a self-reconfiguring platform

Fatma Abouelella, Karel Bruneel and Dirk Stroobandt
Ghent University, ELIS Department
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{fatma.abouelella, Karel.Bruneel, Dirk.Stroobandt}@UGent.be

Abstract—The inherent reconfigurability of SRAM-based FPGAs enables the use of configurations optimized for the problem at hand. Optimized configurations are smaller and faster than their generic counterparts and therefore use the FPGAs resources more efficiently. However, when the problem at hand changes, two overheads are introduced: generating the new configuration and reconfiguring the FPGA. Many authors have tried to reduce these overheads with various success. The most successful implementations are hand designs for one specific application. The methods used in these implementations are hard to port to other applications, which results in a big design cost. To make run-time reconfiguration feasible in commercial designs, automated design methods are needed. In this paper, we describe a tool flow that automatically maps an application to a self-reconfiguring platform. This platform contains a configuration manager, responsible for the reconfiguration and an FPGA. In our case the Power PC (PPC) of a Xilinx FPGA is used as configuration manager. It uses the FPGA's ICAP to reconfigure the FPGA. The tool flow is build around a reconfigurability-aware technology mapper called TMAP [2]. TMAP is an extension of a conventional technology mapper that produces a Tunable LUT circuit, i.e., a LUT circuit in which some of the LUT truth tables are expressed as functions of a set of parameters. The input of the tool flow is an annotated HDL design, that describes the functionality that needs to be implemented. The flow outputs an FPGA configuration and a set of C functions. The FPGA configuration is used to configure the FPGA at startup and the C-functions are used by the PPC to adapt this configuration at run-time. The approach is successfully used to automatically implement an adaptive filter on the Xilinx XUP board.

Keywords—CAD, FPGA, run-time reconfiguration, ICAP

I. INTRODUCTION

The inherent reconfigurability of SRAM-based FPGAs enables the use of several different implementations of the same application, each optimized for the problem at hand at different time intervals. Each implementation is called a configuration. Optimized configurations are smaller and faster than their generic counterparts and therefore use the FPGA's resources more efficiently. However, at the time interval boundaries, the problem at hand will change

and valuable system resources need to be used to generate/select a new configuration and reconfigure the system's FPGA.

Conventional synthesis tools generate FPGA configurations from scratch. They use heuristics to solve NP-complete problems like placement and routing. Hence, generating a configuration requires huge amounts of resources. This makes run-time generation of configurations with conventional tools inefficient.

The authors of [2] have shown that it is possible, for a large set of problems, to generate a new configuration with significantly less resources without sacrificing quality, making run-time reconfiguration feasible. Indeed, in many applications, subsequent data manipulations only differ in a small set of parameter values. This property enables an off-line generation process that results in an FPGA configuration where some of the configuration bits are expressed as Boolean functions of the parameters. On-line specialization then means evaluating these functions. One can see that the number of resources needed to evaluate closed form Boolean functions is much smaller than the resources needed by conventional synthesis tools.

To make run-time reconfiguration feasible in commercial designs, automated design methods are needed. In this paper, we describe a tool flow that automatically maps an application to a self-reconfiguring platform, thus removing all impediments for an easy adoption of run-time reconfiguration in commercial designs.

We have used our new tool flow to automatically implement an adaptive 32-tap FIR filter on a Xilinx XUP board with a Virtex-II Pro device. Our new tool flow results in a 40% reduction in overall FPGA area usage while the reconfiguration overhead is kept manageable.

Our paper starts with an overview of the run-time reconfiguration methodology in Section II. Section III explains the fully automatic tool flow that enables the commercial use of run-time reconfiguration. Although a completely new tool flow could be developed that is ideally targeted at our run-time reconfiguration methodology, we have implemented the generic tool flow in a commercial Xilinx EDK-

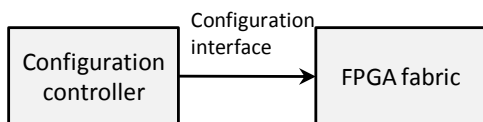


Fig. 1. Self-reconfiguring platform.

based tool flow in which we have used as many standardly available tools as possible. Finally, the experimental results are described in Section IV.

II. PARAMETERIZABLE RUN-TIME RECONFIGURATION METHODOLOGY

The basis of our reconfiguration methodology is the notion that any reconfiguration process has to change the values stored in the configuration memory bits of an FPGA fabric. These configuration memory bits control which wires are connected and what functionality is implemented by a particular piece of logic. As shown in Fig. 1, a configuration controller [3] sends the configuration data to the configuration interface of the FPGA fabric. The controller is responsible for creating the configurations based on the circuit implementation and the architecture of the FPGA and configuring the FPGA with the new configuration data.

Conventional synthesis tools generate FPGA configurations from scratch. This involves using heuristics to solve NP-complete problems like placement and routing. Hence, generating a new configuration requires huge amounts of configuration controller resources, making run-time generation of configurations with conventional tools generally inefficient. However, in many applications, subsequent data manipulations only differ in a small set of parameter values. In between these parameter changes, the parameter set remains at constant values during relatively long time intervals. This property enables us to do much of the generation process off-line, resulting in a *parameterizable configuration* that makes use of the fact that parameters remain constant at each configuration. In such a FPGA configuration, some of the configuration bits are expressed as closed form Boolean functions of a set of parameters, called the *tuning functions*. On-line specialization of a parameterizable configuration means evaluating these tuning functions. One can easily see that the number of resources needed to evaluate closed form Boolean functions is much smaller than the resources needed by conventional synthesis tools. This method hence allows a designer to optimize a design in each time interval in between two parameter changes but without the need for an infeasibly long generation time.

In [2], the authors presented a generic method for automatically generating run-time parameterizable configu-

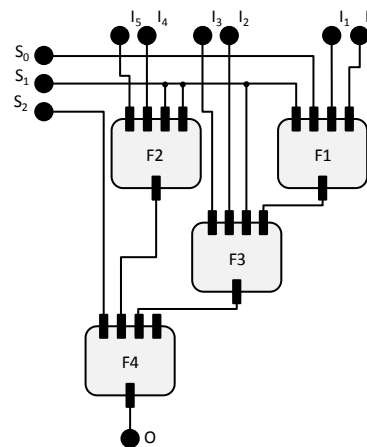


Fig. 2. LUT circuit and boolean functions of the 6:1 multiplexer example.

rations for LUT-based FPGAs. The method starts from an arbitrary digital circuit and a set of parameter inputs. These parameter inputs are a subset of the circuit's inputs. Their mapping method, developed on the basis of the concept of parameterizable configurations, can map an arbitrary Boolean circuit to a parameterizable configuration. The core of this method is TMAP, an extension of conventional technology mapping that produces a Tunable LUT (TLUT) circuit, i.e., a LUT circuit in which some of the LUT truth tables are expressed as function of a set of parameters. We will explain this concept on the example of a 6:1 multiplexer. The multiplexer has six data inputs (I_0, I_1, I_2, I_3, I_4 and I_5), three select inputs (S_0, S_1 and S_2) and one output (O). The circuit shown in Fig. 2 is the multiplexer circuit obtained by conventional synthesis and mapping tools.

In the parameterizable reconfiguration methodology, inputs S_0, S_1 and S_2 are selected to be parameter inputs. As these inputs remain constant in each time interval between two parameter changes, they do not have to be synthesised as real inputs but they can be included in the LUT function. Therefore, conceptually, TMAP generates the TLUT circuit of Fig 3. A TLUT has up to four regular inputs (just as a regular LUT) and any number of parameter inputs. Since we have 6 regular inputs and 3 parameter inputs, we only need two TLUTs instead of four LUTs in the original circuit, a significant reduction. Assuming LUTs in the target FPGA fabric have four inputs each, there are 16 tuning functions associated to each TLUT which express the functionality of the TLUT depending on the parameter inputs.

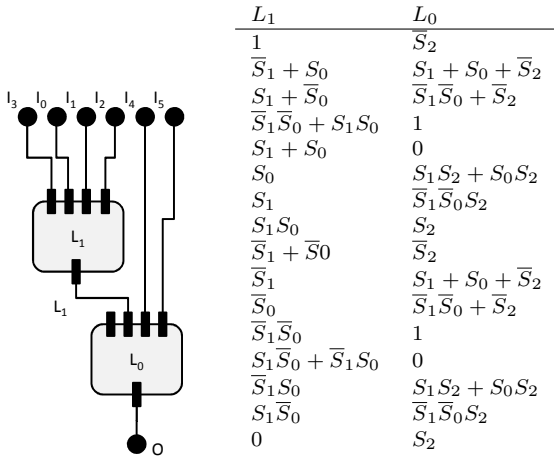


Fig. 3. TLUT circuit of the 6:1 multiplexer example and corresponding tuning functions. The parameter inputs are removed from the figure, resulting in a static LUT circuit.

III. RUN-TIME RECONFIGURATION TOOL FLOW

The run-time reconfiguration methodology explained in the previous section can only be commercially viable if an automatic tool flow relieves the designer of the burden to introduce the reconfiguration possibility. We introduce such an automatic tool flow in Fig. 4. It takes a parameterizable HDL design as input, i.e., a design in which the rate of change in the values of some inputs, called the *parameters*, is considerably lower than that of the other inputs.

After a conventional synthesis step, the TMAP method maps the design into a TLUT circuit and generates the tuning functions associated to each TLUT. Obtaining the static LUT circuit can simply be done by removing the parameter inputs and the associated connections from this TLUT circuit, as was done in Fig. 3. The static LUT circuit can then be implemented using conventional placement and routing tools. The output of the implementation process is the placed and routed circuit, which is used to generate an initial bit stream to configure the FPGA at start-up. For the subsequent reconfigurations, we need to reconfigure the correct LUT bits to change the LUT's function according to the new parameter values. For this, we need to know where these bits are mapped in the configuration memory. These locations can easily be extracted from the information generated by the place and route tools. Once the exact location of the bits to be reconfigured is found, this information is used to generate C code that evaluates the tuning functions and reconfigures the LUTs.

Next to the reconfigurable implementation, the tool flow also produces a simulation model. This model is an HDL design based on the TLUT circuit generated with TMAP. It can be simulated to check the reconfigurable design against the original HDL design functionality with con-

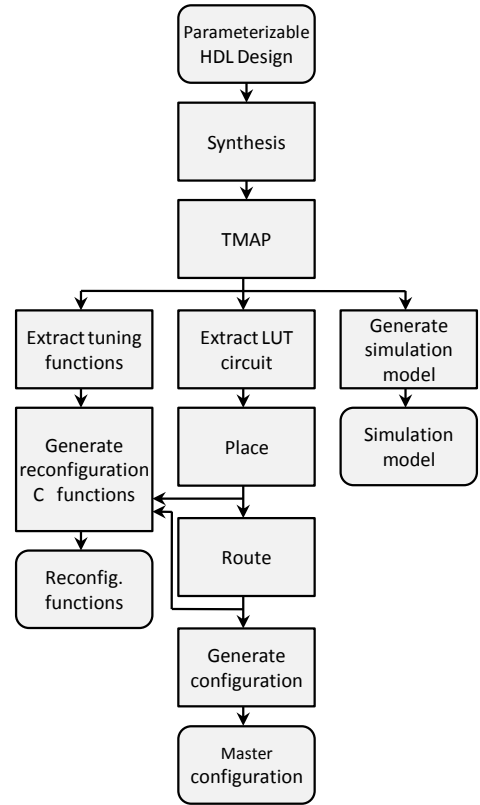


Fig. 4. Our automatic run-time reconfiguration tool flow.

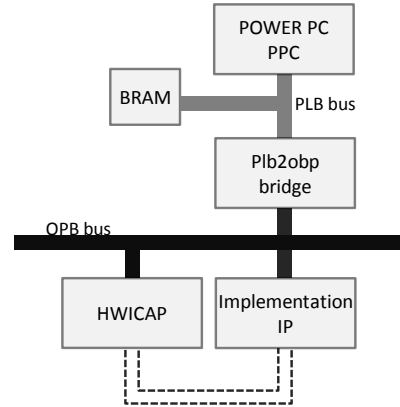


Fig. 5. EDK platform

ventional simulation tools.

IV. EXPERIMENTS AND RESULTS

Our practical tool flow implementation uses Xilinx components and a Xilinx tool flow and is based on the general tool flow of Fig. 4 and the self-reconfiguration platform of Fig. 1. The configuration controller is implemented on an embedded Power PC (PPC) of the Xilinx Virtex-II pro FPGA, which ensures a tight connection to the Virtex-II pro FPGA fabric [1]. To configure parts of the FPGA fabric (LUTs) after a parameter value has changed, the PPC

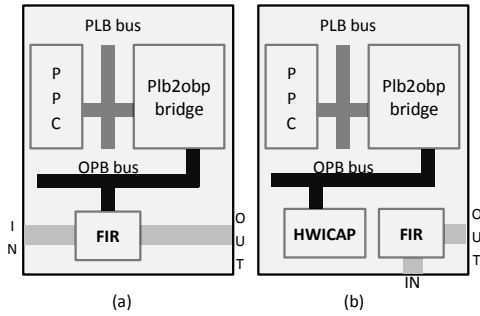


Fig. 6. (a) Block diagram of a conventional FIR filter implementation. (b) Block diagram of the reconfigurable implementation.

automatically evaluates the tuning functions, generates the new configuration, and sends this new configuration to the FPGA configuration memory through the ICAP port of the FPGA through the HWICAP module (Fig. 5).

We validate our run-time reconfiguration tool flow on an adaptive filtering application. It implements a 32-tap FIR filter with 8-bit coefficients and an 8-bit input in a fully pipelined way. In this case, we assume that the coefficients need to be changed every once in a while, which could, e.g., be the case in a wifi application to cancel inter-symbol interference (ISI). Every time a wifi-client is moved, the communication channel properties change and the coefficients of the ISI cancelation filter need to be updated. We also assume that the configuration manager is responsible for calculating the new coefficients and reconfiguring the filter accordingly.

We implemented this system on a Xilinx XUP board in two different ways. The first implementation uses a conventional tool flow where the coefficient manager is implemented in software that runs on the FPGA's embedded PowerPC (PPC). New coefficient values can be handled by the filter because it is implemented in the FPGA fabric with generic multipliers. The coefficients are stored in registers which are mapped in the PPC's memory through the PLB and OPB buses. Figure 6 (a) shows a detailed schematic of this first implementation.

The second implementation uses our new toolflow to generate a reconfigurable FIR filter. Again, the coefficient manager is implemented in software, but now the coefficient manager changes the filter characteristics by reconfiguring the FPGA through the ICAP. Figure 6 (b) shows a detailed schematic of the second implementation.

In both implementations the PPC is clocked at 100 MHz and the busses are clocked at 50 MHz. The resource usage of both implementations can be found in Table I. The table shows that the reconfigurable implementation requires 2,194 (40%) less LUTs to implement the adaptive filter.

TABLE I
COMPARISON OF THE CONVENTIONAL IMPLEMENTATION
AND THE RECONFIGURABLE IMPLEMENTATION.

	Conventional	Reconfigurable
FIR Area (LUTs)	4,259	1,985
System Area (LUTs)	1,182	1,298
Total Area (LUTs)	5,477	3,283

This is mainly because of the size reduction of the run-time reconfigurable FIR filter versus the generic FIR filter. The coefficient controller is only slightly bigger in the reconfigurable implementation because of the additional HWICAP module that is needed to connect the PPC to the ICAP.

Of course, this size reduction does not come for free. The total time needed to change the coefficients in the reconfigurable implementation is 215 msec, which is much larger than for the conventional implementation, which requires only a few clock cycles to change the coefficients. However, this is not an infeasible overhead since the intended system does not require rapid changes of the coefficients and the (optimized) system runs a lot longer in between two coefficient changes.

V. CONCLUSION

Run-time hardware reconfiguration provides ample opportunities for optimizations of an implementation in time intervals in between two parameter changes. This paper provides a general tool flow that automatically maps any application that has a set of slowly varying inputs (called the parameters) to a self-reconfigurable platform. We have effectively implemented the tool flow in a Xilinx EDK platform on a Virtex-II Pro FPGA device, using the embedded power PC as reconfiguration manager. Experimental results on a 32-tap adaptive filter show that the use of self-reconfiguration with our tool flow improves the resource demands of the application by 40% without introducing a prohibitively large reconfiguration generation overhead. Yet, the optimization possibilities at each time interval remain fully flexible as each combination of parameter values can be handled at any moment of the implementation run time.

REFERENCES

- [1] B. Blodget, P. James-Roxby, E. Kelle, S. McMillan, and P. Sundararajan. A selfreconfiguring platform. *FPL*, 2003.
- [2] K. Bruneel and D. Stroobandt. Automatic generation of run-time parameterizable configurations. *FPL*, 2008.
- [3] S. Häuck and A. Dehon, editors. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Systems on Silicon. Morgan Kaufmann, November 2007.