Automatically Mapping Applications to a Self-reconfiguring Platform

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

***Abstract*—The inherent reconfigurability of FPGAs enables us to optimize an FPGA implementation in different time intervals by generating new optimized FPGA configurations and reconfiguring the FPGA at the interval boundaries. With conventional methods, generating a configuration at run-time requires an unacceptable amount of resources. In this paper, we describe a tool flow that can automatically map a large set of applications to a self-reconfiguring platform, without an excessive need for resources at run-time. The self-reconfiguring platform is implemented on a Xilinx Virtex-II Pro FPGA and uses the FPGA's PowerPC as configuration manager. This configuration manager generates optimized configurations on-the-fly and writes them to the configuration memory using the ICAP. We successfully used our approach to implement an adaptive 32-tap FIR filter on a Xilinx XUP board. This resulted in a 40% reduction in FPGA resources compared to a conventional implementation and a manageable reconfiguration overhead.**

I. Introduction

The inherent reconfigurability of SRAM-based FPGAs enables the use of different configurations at different time intervals, each optimized for the specific task in the corresponding time interval. Optimized configurations are smaller and faster than their generic counterparts and require less power. Therefore, they use the FPGA's resources more efficiently. However, at the time interval boundaries, the problem at hand will change and valuable resources will need to be used to generate or select a new configuration and reconfigure the FPGA.

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

Several authors have tried to reduce the resources needed for generating configurations. One of the strategies has been to use lean versions of conventional tools [1], [2]. These lean tools trade quality of the configuration for resources. However, the reduction in resources is limited and often the reduced quality of the configuration negates the overall performance gain. If the number of possible configurations is limited, good quality configurations can be generated off-line and stored in memory [3], [4]. This significantly decreases the needed resources because the on-line generation step is reduced to selecting the right configuration from the memory. However, a very large number of different configurations is generally needed, rendering this technique often infeasible. This last problem can be addressed by generating a generic implementation off-line and rapidly specializing this implementation at run-time [5], [6], [7], [8]. Although these methods reduce the need for resources, the reduction of the configuration quality is mostly inacceptable.

The strategies described up till now are generic. Others have focused on one specific application, e.g., multiplication [9]. These approaches show the best reduction in reconfiguration resources, but they require huge amounts of design time because the design has to be done at the LUT level.

In [10], we have shown that it is possible, for a large set of problems, to generate an arbitrary new configuration with significantly less resources and without sacrificing much quality. Indeed, 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 an off-line generation process that results in an FPGA configuration where some of the configuration bits are expressed as closed form Boolean functions of these parameters, called *parameterizable configurations*. On-line specialization then means evaluating these 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 unfeasibly long generation time.

However, the ability to provide an efficient run-time reconfiguration solution is not sufficient. 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 a high-level description of 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, in Section II, of a method to generate efficient parameterizable configurations.

978-3-9810801-5-5/DATE09 © 2009 EDAA

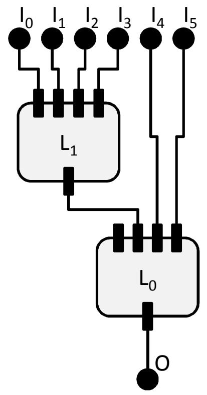
In Section III, we explain how to exploit parameterizable configurations for the run-time reconfiguration mechanism in a self-reconfiguring platform. This generic approach is then further elaborated in Section IV, where we describe a practical tool flow targeting commercially available (Xilinx Virtex-II Pro) FPGAs. Finally, the experimental results are described in Section V.

II. Parameterizable Configurations

The basis of our reconfiguration method is the notion that the bits that form an FPGA configuration can be expressed as a Boolean function of a set of parameters, called *tuning functions*. A configuration in which some of the configuration bits are expressed as tuning functions is called a *parameterizable configuration* [10]. The most important property of such a parameterizable configuration is that it can very rapidly be transformed into a regular configuration for one specific set of parameter values by simply evaluating its tuning functions. One can easily see that compared to the NP-hard placement and routing problems that need to be solved when generating an FPGA configuration in the conventional way, evaluating Boolean functions is a lot less labor intensive. What we actually do is solve the placement and routing problem once, as will be seen in the next paragraph, and reuse this solution every time we generate a regular configuration from the parameterizable configuration.

In [10], we presented a generic method for automatically generating parameterizable configurations for LUT-based FPGAs from an arbitrary parameterizable circuit. Such a circuit contains *parameter inputs* which change values less often than the general inputs. The core of the method is TMAP, a reconfigurability-aware technology mapper that produces a *tunable LUT* (TLUT) circuit, i.e., a LUT circuit in which some of the LUT truth tables are expressed as a function of the parameters. Because the parameters only affect the truth tables and not the TLUT structure, this TLUT circuit can then be placed and routed offline using conventional tools to form a parameterizable configuration.[1] An important property of the parameterizable configurations produced by TMAP is that the specialized regular configurations derived from them generally result in an implementation that uses less area and runs faster than a generic FPGA implementation with the same functionality, while maintaining full flexibility of assigning values to the parameters.

To illustrate the concept of TMAP, we will explain it on the example of a 6:1 multiplexer. The multiplexer has six data inputs (I_0 through I_5), three select inputs (S_0, S_1 and S_2) and one output (O). Without loss of generality, we choose the select inputs S_0, S_1 and S_2 as parameter inputs.[2] One can easily see that a conventional technology mapper will need at least four 4-input LUTs,[3] with fixed LUT truth tables, to implement this multiplexer. This is the generic FPGA implementation.



L_1	L_0
0	0
S_0S_1	$S_0\overline{S}_1S_2$
$\overline{S}_0S_1$	$\overline{S}_0\overline{S}_1S_2$
S_1	$\overline{S}_1S_2$
$S_0\overline{S}_1$	$\overline{S}_2$
S_0	$S_0\overline{S}_1 + \overline{S}_2$
$\overline{S}_0S_1 + S_0\overline{S}_1$	$\overline{S}_0\overline{S}_1 + \overline{S}_2$
$S_0 + S_1$	$\overline{S}_1 + \overline{S}_2$
$\overline{S}_0\overline{S}_1$	0
$S_0S_1 + \overline{S}_0\overline{S}_1$	$S_0\overline{S}_1S_2$
$\overline{S}_0$	$\overline{S}_0\overline{S}_1S_2$
$\overline{S}_0 + S_1$	$\overline{S}_1S_2$
$\overline{S}_1$	$\overline{S}_2$
$S_0 + \overline{S}_1$	$S_0\overline{S}_1 + \overline{S}_2$
$\overline{S}_0 + \overline{S}_1$	$\overline{S}_0\overline{S}_1 + \overline{S}_2$
1	$\overline{S}_1 + \overline{S}_2$

Fig. 1: TLUT circuit of the 6:1 multiplexer example and corresponding tuning functions.

In the case of parameterizable reconfigurations, the parameter inputs (the multiplexer's selection inputs in our example) can be included in the LUT function and do not have to be synthesized as real inputs. In this case, TMAP can implement the multiplexer using only two TLUTs, as shown in Fig. 1, a significant reduction of 50% in area for this example. Because we assume the target FPGA fabric has 4-input LUTs, there are 16 tuning functions associated to each TLUT's truth table, which express the functionality of the TLUT depending on the parameter inputs. If the parameters S_0, S_1, and S_2 equal, e.g., 1, 0 and 1 respectively, the tuning functions of LUTs L_0 and L_1 respectively evaluate to the regular configuration vectors 0101010101010101 and 0000111100001111.

III. Mapping Parameterizable Applications to a Self-reconfiguring Platform

The fact that parameterizable configurations can rapidly be transformed into specialized FPGA configurations and the observation that many applications contain parameters that remain at constant values during relatively long time intervals enables us to use the concept of parameterizable configurations in self-reconfiguring systems. Upon the change of a parameter value at run-time, such a system will evaluate the tuning functions in order to obtain new configuration bit values and will then reconfigure its FPGA fabric using these values. We call the subsystem responsible for these two tasks the *configuration manager*. In this paper, we assume the configuration manager is a process running on an instruction set processor.

The tool flow used to map a parameterizable HDL design to a self-reconfiguring platform is shown in Fig. 2. It actually does not produce a parameterizable configuration directly. Instead, it produces both a master configuration, which is used

[1]Note that in this case only the LUT truth table bits can be expressed as a function of the parameters. We are currently also investigating solutions that can touch the other reconfiguration bits efficiently.

[2]Up to now, the selection depends on the designer's experience but in the future we want to also make parameter selection automatic.

[3]In this paper, we always assume a LUT has four inputs. Our tool flow can also be used for the newer six-input LUTs but for the sake of clarity we only describe the four-input case here.

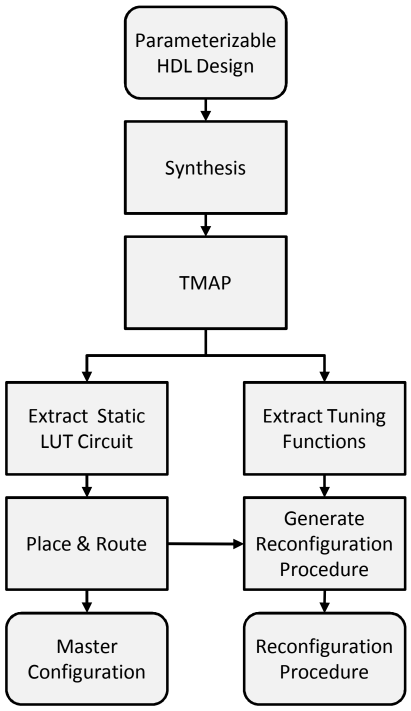

Parameterizable HDL Design
Synthesis
TMAP
Extract Static LUT Circuit
Extract Tuning Functions
Place & Route
Generate Reconfiguration Procedure
Master Configuration
Reconfiguration Procedure

Fig. 2: Tool flow for mapping a parameterizable HDL design to a self-reconfiguring platform.

to generate an initial bit stream to configure the FPGA at start-up, and a set of specialized reconfiguration functions, that serve as the basis for the configuration manager. The configuration manager is thus specialized for the application at hand. This reduces the resource needs compared to a generic configuration manager that takes any parameterizable configuration and the set of parameter values to produce a new configuration.

After a conventional synthesis step, TMAP maps the design into a TLUT circuit. Obtaining the static LUT circuit can simply be done by ignoring that the TLUT truth tables depend on the parameter values. The static LUT circuit can then be implemented using conventional placement and routing tools. The output of this implementation process is the master configuration.

In order to change one specific TLUT's function according to new parameter values, we need to know the way its truth table is related to the parameter values, i.e., the tuning functions, and the location of the physical LUT that implements the TLUT under consideration. The combination of this information for all TLUTs is used to generate a C procedure that reconfigures the FPGA. This is shown in the right branch of the tool flow of Fig. 2. The arguments of the procedure are the parameter values.

The fact that we model reconfiguration as a change of parameter inputs which can be expressed in the form of a high level HDL design and that we provide a method that automatically maps this design to a self-reconfiguring platform greatly relieves the designer of the burden to exploit the reconfiguration possibility at the LUT level.

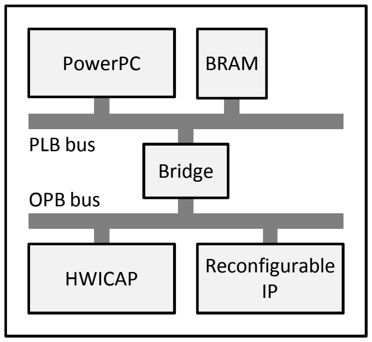


Fig. 3: Self-reconfiguring platform implemented on a Xilinx Virtex-II Pro FPGA.

IV. Practical Tool Flow Instance

In the previous section, we presented the general tool flow to map an application to a self-reconfiguring platform. However, to enable a commercial introduction of this tool flow without too many hurdles, we have searched for a practical tool flow that uses current commercial tools as much as possible and only needs a very limited amount of additional tools. The tool flow presented in this section targets Xilinx components and reuses many Xilinx tools.

The self-reconfiguring platform (Fig. 3) targeted by our tool flow is implemented on a Xilinx Virtex-II Pro FPGA. The configuration manager is implemented on an embedded PowerPC (PPC) of the Xilinx Virtex-II Pro FPGA, which ensures a tight connection to the FPGA fabric [11]. The connection between the configuration manager and the configuration memory is realized through the Xilinx HWICAP module, which provides the interface between the OPB bus and the FPGA's ICAP (Internal Configuration Access Port). To configure parts of the FPGA fabric (LUTs) after a parameter value has changed, the PPC 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 via the HWICAP module. The entire reconfiguration flow is thus executed within the system and no external source is needed to reconfigure the FPGA, nor to take the decision to reconfigure. Therefore, this system is a true *self-reconfiguring* system.

The self-reconfiguring platform shown in Fig. 3 is implemented using Xilinx XPS [12]. The XPS tool flow is implemented in a makefile and it is therefore easy to insert our tools in the flow. The adapted tool flow is shown in Fig. 4.

A. Generating the Master Configuration

We assume that the parameterizable HDL design contains a number of parameterizable modules and a number of non-parameterizable modules. A parameterizable VHDL module is nothing more than a regular VHDL description with annotations indicating which of the inputs are the parameter inputs. The parameterizable module of the 6:1 multiplexer example is shown in Fig. 5. The annotation `--PARAM` indicates that

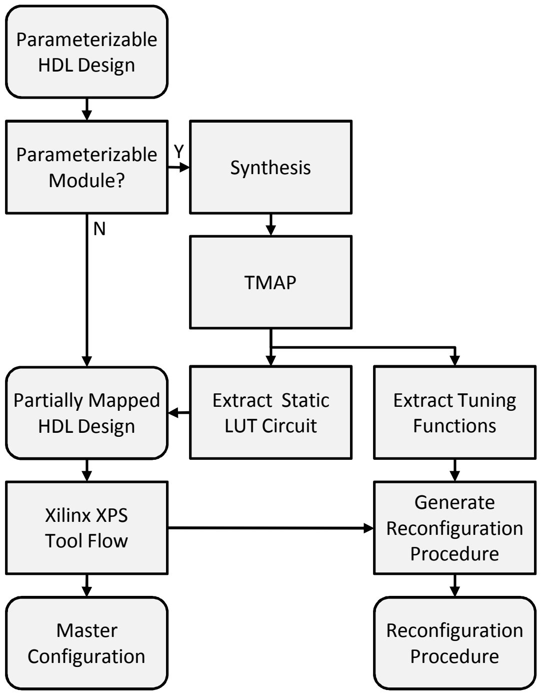

Parameterizable HDL Design
Parameterizable Module?
Y
N
Synthesis
TMAP
Partially Mapped HDL Design
Extract Static LUT Circuit
Extract Tuning Functions
Xilinx XPS Tool Flow
Generate Reconfiguration Procedure
Master Configuration
Reconfiguration Procedure

Fig. 4: Practical tool flow for mapping a parameterizable HDL design to a self-reconfiguring platform.

```
entity mux6 is
port(
  s : in  std_logic_vector(2 downto 0); --PARAM
  i : in  std_logic_vector(5 downto 0);
  o : out std_logic);
end mux6;

architecture behavior of mux6 is
begin
  o <= i(conv_integer(s));
end behavior;
```

Fig. 5: Parameterizable VHDL module of the 6:1 multiplexer example.

the select inputs are parameters. As the annotations are in a comment line, any conventional synthesis tool can be used to synthesize the circuit. We used Altera Quartus II because it can dump a .blif file that can then be used as input for our mapper TMAP [10], which maps the circuit to a TLUT circuit.

We make a distinction between parameterizable modules and non-parameterizable modules. Indeed, the Virtex-II Pro architecture is a very heterogeneous architecture compared to the homogeneous LUT architecture that TMAP targets. Therefore, using TMAP to map the full design would result in a very inefficient use of the Virtex-II Pro architecture. We thus limit the use of TMAP to the parameterizable modules, as is shown in Fig. 4. The static LUT circuit of these modules is expressed in VHDL by directly instantiating LUTs in the VHDL module. Combining these modules with the non-parameterizable VHDL modules of the original design forms the partially mapped HDL design. This VHDL design can now be efficiently mapped to the Virtex-II Pro architecture by the Xilinx tools without corrupting the mapping done by TMAP. The result of this last mapping is the master configuration. This workaround could of course be avoided if the ability to map to TLUTs would be incorporated in the Xilinx mapper.

It is important to note that every LUT instantiated in VHDL is given a unique name. This enables our tools to find the LUT's location after place and route, see Section IV-B. Although it is not strictly necessary, we also lock the pins of the LUTs with the `lock_pins` attribute so that the router does not interchange the pins during routing. This greatly simplifies generating the reconfiguration procedure.

B. Generating the Reconfiguration Procedure

The reconfiguration procedure reconfigures all the TLUTs instantiated in a parameterizable module according to the parameter values that are passed as arguments to the procedure.

As mentioned in Section III, we need both the tuning functions of each TLUT and the location of each TLUT in order to do the reconfiguration upon a parameter change. The tuning functions for each TLUT are provided by TMAP, this is explained in detail in [10]. The LUT locations are harder to come by. On the Virtex-II Pro a LUT location is specified by a slice row, a slice column and whether it's the F or the G LUT of the slice [13]. Finding these locations for each instantiated LUT is done in the following way. The Xilinx tool flow generates a .NCD file that contains all the information on the mapped circuit including the location of the LUTs. This .NCD file is first converted to a .XDL file, a clear-text representation of the .NCD file, using the Xilinx XDL program [14]. We find the LUT locations in this .XDL file by searching the unique names given to the LUTs when they were instantiated in VHDL, as explained in Section IV-A.

A reconfiguration procedure is then generated as follows. For each of the TLUTs in a parameterizable module we generate a TLUT reconfiguration procedure that takes the module parameter values as inputs, evaluates the tuning functions generated by TMAP and reconfigures the LUT. The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example is shown in Fig. 6. The code that evaluates the tuning functions of a TLUT is generated by simply translating the expressions produced by TMAP into C-style expressions.[4] When executed, these expressions result in a new truth table for the LUT. The reconfiguration of the LUT is then done by calling the procedure `XHwIcap_SetClbBits`, which is provided by Xilinx in the HWICAP module driver. This procedure takes the LUT location and the new truth table to reconfigure the LUT. In our example we assume that LUT L_1 is located in the G LUT of the slice at row 31 and column 45. The reconfiguration procedure for a module simply calls the TLUT reconfiguration procedure for each of the TLUTs of a module. The reconfiguration procedure for our 6:1 multiplexer example is shown in Fig. 7.

[4]It must be noted that, since the Virtex-II Pro family LUT configurations are stored in an inverted way, the configuration data must be inverted before configuring the LUTs [15].

```
void L1( XHwIcap *hwIcap,
         Xuint8 S0, Xuint8 S1, Xuint8 S2) {

  Xuint8 truthTable[LUT_SIZE];
  truthTable [0] = !(0);
  truthTable [1] = !(S0 && S1);
  truthTable [2] = !(!S0 && S1);
  truthTable [3] = !(S1);
  truthTable [4] = !(S0 && !S1);
  truthTable [5] = !(S0);
  truthTable [6] = !((!S0 && S1) || (S0 && !S1));
  truthTable [7] = !( S0 || S1);
  truthTable [8] = !(!S0 && !S1);
  truthTable [9] = !((S0 && S1) || (!S0 && !S1));
  truthTable [10]= !(!S0);
  truthTable [11]= !(!S0 || S1);
  truthTable [12]= !(!S1);
  truthTable [13]= !(S0 || !S1);
  truthTable [14]= !(!S0 || !S1);
  truthTable [15]= !(1);

  XHwIcap_SetClbBits( hwIcap, 31, 45, G_LUT,
                      truthTable, LUT_SIZE);
}
```

Fig. 6: The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example. We assume that LUT L_1 is located in the G LUT of the slice at row 31 and column 45.

```
void mux2w1 ( XHwIcap *hwIcap,
              Xuint8 S0, Xuint8 S1, Xuint8 S2) {
  L0(hwIcap, S0, S1, S2);
  L1(hwIcap, S0, S1, S2);
}
```

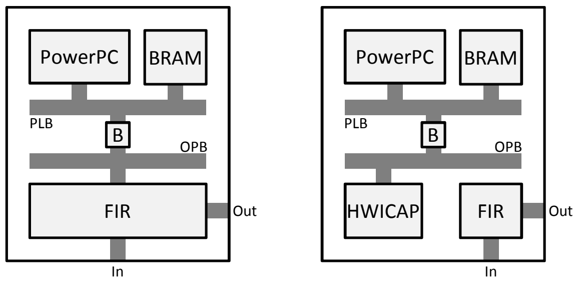
Fig. 7: The reconfiguration procedure for our 6:1 multiplexer example.

On a last practical note, we should warn the reader that, in the Virtex-II Pro family, reconfiguring a LUT will cause corrupted data in the SRL16s and LUT RAMs that are located in the same column. Therefore, placing TLUTs in the same columns as SRL16s or LUT RAMs must be avoided. This can be done using `AREA_GROUP` constraints. This is no longer an issue in the Virtex-5 family.

V. Experiments and Results

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. 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 way is the conventional way. The filter is implemented using generic multipliers and coefficient values are handled as regular inputs to the filter. The coefficients are stored in registers which are mapped in the PPC's memory through the PLB and OPB buses. The coefficient manager is implemented as software on the PPC. It changes the filter characteristics by writing registers. Figure 8 (a) shows a detailed schematic of this first implementation.



(a) (b)

Fig. 8: (a) Block diagram of a conventional adaptive FIR filter implementation. (b) Block diagram of the reconfigurable FIR filter implementation.

TABLE I: Comparison of the conventional implementation and the reconfigurable implementation.

	Conventional	Reconfigurable
FIR Area (LUTs)	4,259	1,985
System Area (LUTs)	1,218	1,298
Total Area (LUTs)	5,477	3,283
Reconf. time (ms)	N/A	151

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, as explained in Section IV. Figure 8 (b) shows a detailed schematic of the second implementation.

In both implementations the PPC is clocked at 200 MHz and the busses are clocked at 66 MHz.[5] The resource usage of both implementations can be found in Table I. The table shows that the reconfigurable implementation requires in total 2,194 (40%) less LUTs to implement the adaptive filter. This is mainly because of the size reduction (by over 53%) 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 HW-ICAP module that is needed to connect the PPC to the ICAP, and the memory needed to store the tuning functions which occupies 30% of the total memory of the PowerPC. The goal of this paper is mainly to show the concept of our automatic reconfiguration tool flow and to show its benefits on a single example implementation. We envisage the implementation of more elaborate examples in the future to also show the benefits of the tool flow for much larger designs. The first results indeed look promising.

Of course, this size reduction does not come for free. The

[5]The maximal clock frequency of the ICAP port in the Virtex-II Pro family is 66 MHz.

time needed to change the coefficients in the reconfigurable implementation, 151 ms, 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. In the future, we want to further optimize the reconfiguration process and thus even reduce the reconfiguration time. At this moment, reconfiguring is done one LUT at a time. Since the reconfiguration atom of a Virtex-II Pro FPGA spans a full column of LUTs, reconfiguring a full column at a time will significantly reduce the reconfiguration time.

Also the offline (static) overhead of our reconfiguration method is low. The additional tool flow time for enabling the reconfiguration[6] is about 17% of the total time needed (11 minutes in this experiment) to implement the reconfigurable implementation in XPS. This overhead could be further reduced by optimizing the generation of the reconfiguration procedure.

VI. 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 integrated our tool flow in the Xilinx XPS tool flow that targets Virtex-II Pro FPGA devices. We used the embedded PowerPC of the Virtex-II Pro device 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 optimized design remains fully flexible as each combination of parameter values can be handled at any moment of the implementation run time.

[6]TMAP time and the time to extract the tuning functions and generate the reconfiguration functions.

References

[1] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," *Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, July 2006.

[2] Y. Sankar and J. Rose, "Trading quality for compile time: ultra-fast placement for FPGAs," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1999, pp. 157–166.

[3] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," vol. 6, no. 2, pp. 247–256, 1998.

[4] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable computing solutions for automatic target recognition," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1996.

[5] J. Leonard and W. H. Mangione-Smith, "A case study of partially evaluated hardware circuits: Key-specific DES," in *Proc. International Workshop on Field-Programmable Logic and Applications (FPL)*, 1997, pp. 151–160.

[6] S. Singh, J. Hogg, and D. McAuley, "Expressing dynamic reconfiguration by partial evaluation," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1996.

[7] N. McKay and S. Singh, "Dynamic specialisation of XC6200 FPGAs by partial evaluation," *Lecture Notes in Computer Science*, vol. 1482, p. 298, 1998.

[8] K. Bruneel, P. Bertels, and D. Stroobandt, "A method for fast hardware specialization at run-time," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 35–40.

[9] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," *J. VLSI Signal Process. Syst.*, vol. 36, no. 1, pp. 7–15, 2004.

[10] K. Bruneel and D. Stroobandt, "Automatic generation of run-time parameterizable configurations," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2008, pp. 361–366.

[11] B. Blodget, P. James-Roxby, E. Kelle, S. McMillan, and P. Sundararajan, "A selfreconfiguring platform," *International Conference on Field-Programmable Logic and Applications*, pp. 565– 574, 2003.

[12] *Embedded System Tools Reference Manual*, Xilinx.

[13] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Xilinx.

[14] J.-B. Note and Éric Rannaud, "From the bitstream to the netlist," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 264–264.

[15] A. Upegui and E. Sanchez, "Evolving hardware by dynamically reconfiguring Xilinx FPGAs," in *Evolvable Systems: From Biology to Hardware*, ser. LNCS, J. M. et al., Ed., vol. 3637. Berlin Heidelberg: Springer-Verlag, 2005, pp. 56–65.