An automatic tool flow for the combined implementation of multi-mode circuits

Brahim Al Farisi, Karel Bruneel, João M. P. Cardoso and Dirk Stroobandt
Ghent University, ELIS Department
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
Brahim.AlFarisi@UGent.be

***Abstract*—A multi-mode circuit implements the functionality of a limited number of circuits, called modes, of which at any given time only one needs to be realised. Using run-time reconfiguration of an FPGA, all the modes can be implemented on the same reconfigurable region, requiring only an area that can contain the biggest mode. Typically, conventional run-time reconfiguration techniques generate a configuration for every mode separately. To switch between modes the complete reconfigurable region is rewritten, which often leads to very long reconfiguration times. In this paper we present a novel, fully automated tool flow that exploits similarities between the modes and uses Dynamic Circuit Specialization to drastically reduce reconfiguration time. Experimental results show that the number of bits that is rewritten in the configuration memory reduces with a factor from 4.6× to 5.1× without significant performance penalties.**

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. This is called run-time reconfiguration (RTR). Using RTR, global area can be reduced by reusing FPGA resources between circuits.

The conventional way of building RTR systems is called Modular Dynamic Reconfiguration [1]. Using this technique, different designs, or modules, can be implemented on the same FPGA area, called the reconfigurable region. For every module a configuration is generated by implementing it separately in the reconfigurable region. To switch between the different modules during run-time, the complete reconfigurable region is rewritten with the appropriate configuration. Since a whole area needs to be rewritten, this often leads to very long reconfiguration times [2].

A new, more fine-grained, approach to RTR is Dynamic Circuit Specialization [3]. The tool flow for Dynamic Circuit Specialization starts from an HDL description in which slowly varying signals, called parameters, are annotated. From this description a parameterized FPGA configuration is generated. This is a configuration in which most of the bits are static and only some of the bits, called parameterized bits, correspond to Boolean functions of the parameters. To specialize the FPGA for specific parameter values, only the Boolean functions need to be evaluated and rewritten in the configuration memory. Since the number of these bits is limited, reconfiguration time can be reduced drastically [3].

978-3-9815370-0-0/DATE13/

A multi-mode circuit implements the functionality of a limited number of circuits, called mode circuits or modes, of which at any given time only one needs to be realised. Also, different modes will often exhibit much similarity, since the same functional blocks are used to build up the circuit. An example of a multi-mode circuit is a mobile transceiver that supports different communication standards (like 3G and Wi-Fi), but only uses one at any given time. In this case, every mode is a circuit that contains the necessary functions to support the corresponding communication standard. Since the different modes are mutually exclusive in time, hardware sharing techniques can be considered to optimize area, power and execution time.

Using run-time reconfiguration of an FPGA, all the modes of a multi-mode circuit can be implemented on the same FPGA area, requiring only an area that can contain the biggest mode. In this paper we present a new, fully automated flow that exploits similarities between the modes and uses Dynamic Circuit Specialization to reduce reconfiguration time. For typical multi-mode circuits, experimental results show a 4.6× to 5.1× reduction in reconfiguration time compared to Modular Dynamic Reconfiguration. In this flow, we introduce a novel wire-length driven approach for the combined implementation of different mode circuits that clearly outperforms a previously proposed circuit edge matching technique.

Our paper starts with an overview of the RTR techniques considered in this paper in Section II. In Section III, we explain how we exploit similarities between the modes and use Dynamic Circuit Specialization to generate an efficient parameterized configuration for multi-mode circuits. The experiments and results are discussed in Section IV. Finally, we conclude in Section V.

II. Run-time reconfiguration

With run-time reconfiguration (RTR) it is possible to implement different functions, that are not needed at the same time in the system, on the same FPGA area. This area is generally called the reconfigurable region. Whenever one wants to change between these functions a period of time is needed, called the reconfiguration time, to rewrite the configuration memory. The subsystem that performs the reconfiguration is called the reconfiguration manager and is generally implemented in software. In this section we will discuss two techniques that use run-time reconfiguration: Modular Dynamic

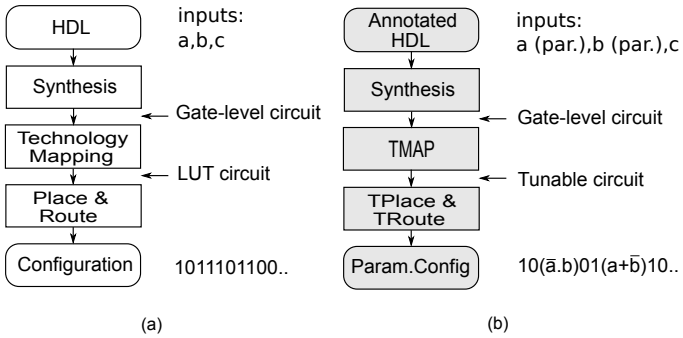

Fig. 1: (a) Conventional FPGA tool flow (b) Dynamic Circuit Specialization (DCS) tool flow

Reconfiguration and Dynamic Circuit Specialization.

A. Modular Dynamic Reconfiguration

The Modular Dynamic Reconfiguration (MDR) tool flow implements every function, also called a module, separately in the reconfigurable region by following the typical steps of the FPGA CAD flow (synthesis, technology mapping, placement and routing) as shown in Figure 1(a). For every module a configuration is generated, that contains the binary values needed to write the configuration memory of the reconfigurable region. To switch between the different modules the reconfiguration manager writes the complete reconfigurable region with the appropriate configuration. Since a whole area needs to be rewritten, this often leads to very long reconfiguration times [1].

When implementing a multi-mode circuit with MDR, every mode will be placed in a separate module.

B. Dynamic Circuit Specialization

Figure 1(b) shows the Dynamic Circuit Specialization (DCS) tool flow compared to the conventional FPGA tool flow. The intermediate representations used in the two flows are also depicted. The DCS tool flow takes in a HDL description in which the slowly varying inputs are annotated as parameters. The tool flow then automatically generates a parameterized configuration, where a limited amount of bits are expressed as Boolean functions of the parameters. When the parameters change value the reconfiguration manager only has to re-evaluate these Boolean functions and write them in the configuration memory.

The synthesis step in the DCS tool flow is very similar as in the conventional tool flow but the technology mapping step is fundamentally different. A conventional technology mapper generates a network of logic blocks, each consisting of a combination of a look-up table and a flip-flop. The truth table entries of the look-up table and the bit that controls the selection of the sequential output are constant zeros and ones. We will further refer to logic blocks simply as look-up tables or LUTs. An example of a 2-input LUT is shown on the left in Figure 4.

The TMAP technology mapper used in DCS, on the other hand, maps the parameterized design onto a Tunable circuit [4]. This is a network of Tunable logic blocks, in short Tunable LUTs or TLUTs, which are logic blocks of which the configuration bits are expressed as a Boolean expression of the parameters. On the right side of Figure 4 an example is shown of a 2-input Tunable LUT. On the bottom side of Figure 3 one can find an example of a Tunable circuit.

Moreover, in contrast with a normal LUT circuit, that contains regular connections, the Tunable LUTs are connected with Tunable connections. These connect a source and a sink, like regular connections, but each Tunable connection is also associated with a Boolean expression that is called the activation function. A Tunable connection only needs to be realised for the parameter values for which the activation function evaluates to True.

TPlace and TRoute, the adapted placer and router in the DCS tool flow, can further refine the Tunable circuit to a parameterized configuration in which some of the configuration bits are expressed as Boolean functions of the parameters [5].

The DCS tool flow provides a way for specializing one design for certain inputs using RTR. It takes one annotated HDL file as input. For multi-mode circuits, however, we want to implement several circuits, each represented by its own HDL description. In the next section we will present an automated tool flow that allows to implement multi-mode circuits using DCS.

III. Generating a parameterized configuration for multi-mode circuits

We first assume the mode circuits are numbered and express this number in a binary fashion. If there are for example 3 modes, we will need 2 bits $m_1 m_0$ to express the mode.

The fully automated tool flow we propose, as shown in Figure 2(b), takes in the HDL descriptions of the different modes and generates a parameterized configuration, in which some of the bits are expressed as a boolean expression of the mode, for example $1, 0, 0, \overline{m_1}.m_0, \overline{m_0}, 1, 0....$ The tool flow is clearly a combination of the MDR and DCS tool flows. The MDR tool flow is followed up until the technology mapping, thus generating a circuit of LUTs for every mode. In the following step the LUT circuits are merged into one Tunable circuit that is further implemented in the reconfigurable region using the TPlace and TRoute step of the DCS tool flow. The proposed tool flow thus reuses much of the steps of the MDR and DCS tool flows. The key step in our tool flow and the main contribution of this paper, is the merging of different LUT circuits into one Tunable circuit, as is shown in Figure 3. It is in this step that we developed novel, automatic techniques that exploit similarities between the modes to reduce reconfiguration time.

Merging of several LUT circuits into a Tunable circuit consists of two steps:

1) determine which LUTs will be implemented using the same Tunable LUT;

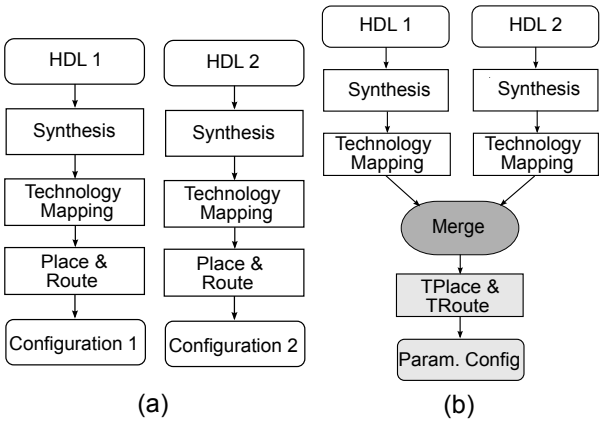

Fig. 2: The tool flow of Modular Dynamic Reconfiguration (a), compared to our approach which uses a merge and Dynamic Circuit Specialization (b).

2) the annotation of the connections with the appropriate activation function to generate the Tunable connections.

Indeed, we essentially have one degree of freedom generating a Tunable circuit, we have to determine which LUTs will be implemented using the same Tunable LUT. Of course, only LUTs belonging to different modes can be combined in the same Tunable LUT. Once this is decided, generating the parameterized bits of the Tunable LUT is very straightforward, as is shown in the example in Figure 4. Every mode circuit corresponds to a Boolean product that evaluates to True for the appropriate mode value. For example, when the mode $m_1 m_0$ is *10* the Boolean product is $m_1.\overline{m_0}$. The bits of a LUT are first multiplied (AND) with the Boolean product of the mode circuit the LUT belongs to. The corresponding bits of the different LUTs are then added (OR) to generate the Boolean expressions that represent the parameterized bits of the Tunable LUT. For example, for the highest bit of the truth table in Figure 4 we get $\overline{m_0}.1 + m_0.0$ which simplifies to $\overline{m_0}$. We note that when evaluating the Tunable LUT on the right for a certain mode value, the correct bit values for the LUTs on the left are obtained. Using the method above, a Tunable LUT can implement any combination of LUTs, as long as they belong to different modes.

The topology of the Tunable circuit is determined, once it is decided which LUTs are implemented in the same Tunable LUT. The connections initially connecting the LUTs will simply connect the corresponding Tunable LUTs. To generate the Tunable connections, the connections of all the modes are annotated with an activation function that consists of the Boolean product that corresponds to the mode circuit the connection belongs to. When connections have the same source and sink they can be merged into one Tunable connection of which the activation function is an addition of the Boolean products of the connections. An example is given in Figure 3. In this figure, connections that are used in both modes have as activation function $\overline{m_0} + m_0$ which simplifies to *True* or *1*. In Figure 3 we simply implement the LUTs with the same

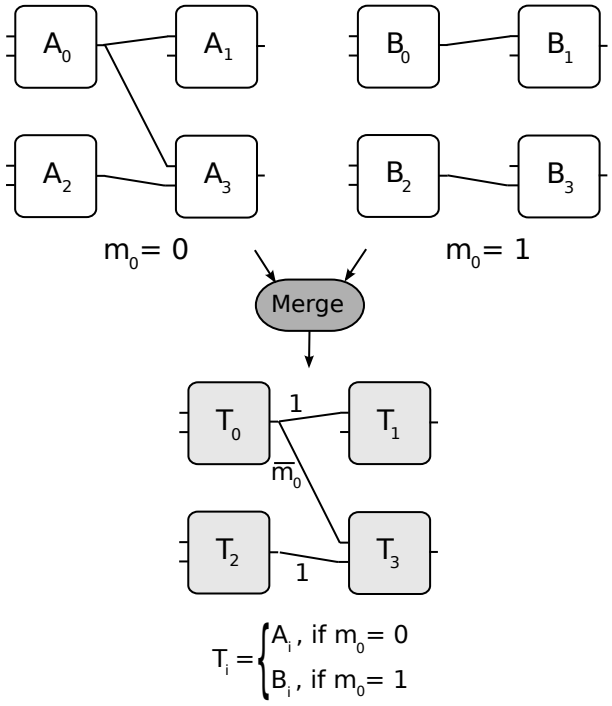

Fig. 3: Merging two LUT circuits into a Tunable circuit.

index using the same Tunable LUT. There are however many ways to combine the different LUTs in one Tunable LUT, each generating a Tunable circuit with a different topology. To obtain an efficient Tunable circuit, we have developed a novel technique, called *combined placement*, which is explained in the following section.

A. Combined placement

Given a placement of all the mode circuits on the reconfigurable region, a Tunable circuit can easily be extracted. The LUTs positioned on the same physical LUT will be implemented using the same Tunable LUT. Using such a combined placement strategy allows to assess both topology and placement quality of the Tunable circuit. In this section we will explain how we extended the conventional placement algorithm to place several LUT circuits simultaneously.

A conventional FPGA placement algorithm takes two inputs: the mapped input circuit and a description of the target FPGA architecture. The algorithm searches a legal placement for the logic blocks of the input circuit so that circuit wiring

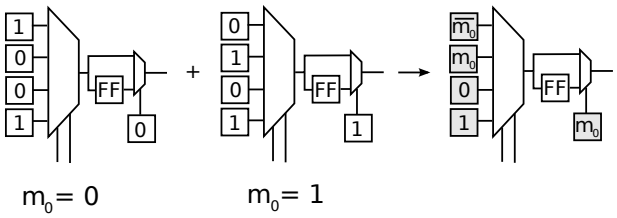

Fig. 4: Generating parametrised Tunable look-up table bits.

is optimised. In a legal placement every LUT is associated to (placed on) one of the physical LUTs (without overlap).

The conventional placement tool is based on simulated annealing. The algorithm starts by randomly, but legally, placing the functional blocks in the input circuit on physical blocks of the FPGA architecture. Afterwards, the placer repeatedly tries to improve the placement cost by interchanging the functional blocks placed on two randomly chosen physical blocks. Such an interchange is called a swap.

We extended the conventional placement tool to accommodate the simultaneous placement of several LUT circuits. First, the LUTs of all the modes are placed randomly on the reconfigurable region. In the conventional placement only one LUT is allowed per physical LUT. In the case of the combined placement, however, LUTs belonging to different modes can be placed on the same physical LUT.

During the combined placement, selecting a swap consists of two steps: choosing two random physical blocks and selecting a mode for which the swap will be executed. Only the LUTs placed on the chosen physical LUTs belonging to the selected mode will be interchanged, the LUTs of the other modes maintain their position.

B. Optimization options

Using the method described in the introduction of III, a Tunable LUT can implement any combination of LUTs as long as they belong to different modes. The approach taken in this paper is to use this degree of freedom to optimize the implementation of the Tunable connections of the Tunable circuit. Indeed, the configuration memory consists mostly of routing bits, thus it is logical to focus on reduction of routing reconfiguration time. In this paper we consider two different approaches to achieve this goal during the combined placement step: *circuit edge matching* and *wire-length optimization*.

The *circuit edge matching* technique tries to reduce the number of Tunable connections by placing the LUTs of the different modes in such a way that the number of connections that have the same source and sink is maximized. As was explained earlier, connections of different modes that have the same source and sink can be merged into one Tunable connection. It is obvious that when one changes between these modes no switches need to be turned in the routing for the merged Tunable connection. Circuit edge matching thus reduces the number of parameterized bits in the routing.

Circuit edge matching was first proposed in [6]. However, these authors did not have a router that could route the wires of the different mode circuits simultaneously. In addition, no results regarding reconfiguration time were presented.

Circuit edge matching only looks at the topology of the Tunable circuit that is formed and does not take into account the placement of the Tunable LUTs. However, using a combined placement strategy the information regarding the placement of the LUTs allows to asses the wire usage of the Tunable circuit. To achieve this goal, the cost function used in the *wire-length optimization* approach uses an estimation of the wire length TRoute will need to route the Tunable circuit. The wire-length estimation used during the combined placement is the same as the one TPlace uses during the placement of the Tunable circuit after merging.

IV. Experiments and Results

A. Benchmarks

To validate our proposed tool flow we conducted experiments using 3 different applications. In the first 2 experiments typical multi-mode applications were used: a regular expression matching (RegExp) and adaptive filtering application (FIR). In [7] a tool was developed that can generate a hardware engine, written in VHDL, that matches a certain regular expression. In the first experiment, we chose 5 regular expressions out of the Bleeding Edge rules set [8] and with this tool generated the corresponding circuits. Then 10 multi-mode circuits were generated by picking all possible combinations of 2 circuits out of the 5 generated circuits. In the second experiment we combined 10 low pass and 10 high pass finite impulse response (FIR) filters into 10 multi-mode circuits. The non-zero coefficients were chosen randomly, after which all the constants were propagated. Such a FIR filter is 3 times smaller than the generic version.

Finally, in the third experiment, we chose 5 circuits out of the general MCNC benchmark suite [9] that were of similar size compared to the rest of the circuits in these experiments. Afterwards we generated 10 multi-mode circuits by making all possible combinations of 2 circuits.

For every set of mode circuits the minimum, average and maximum number of LUTs are reported in Table I.

B. FPGA architecture

The combined placement algorithm was implemented based on our Java version of the VPR (Versatile Place and Route) wire-length driven placer [10]. VPR is the most commonly used academic tool for place and route algorithms. The FPGA architecture used for each of the implementations, is described in `4lut_sanitized.arch`. This is an FPGA architecture file included in the distribution of VPR. It has logic blocks containing one 4-LUT and one flip-flop and the wire segments in the interconnection network only span one logic block. We note that the techniques and tools we use in this paper are independent of the architecture used. The number of inputs of the LUTs is simply an input parameter of the tool flow. Also, different routing architectures can be used since TRoute uses a standard representation of the routing infrastructure called the routing resource graph [10].

Since there is no other functionality implemented on the FPGA, the reconfigurable region comprises the complete

TABLE I: Size of the LUT circuits used in the experiments.

	Minimum	Average	Maximum
RegExp	224	243	261
FIR	235	302	371
MCNC	264	310	404

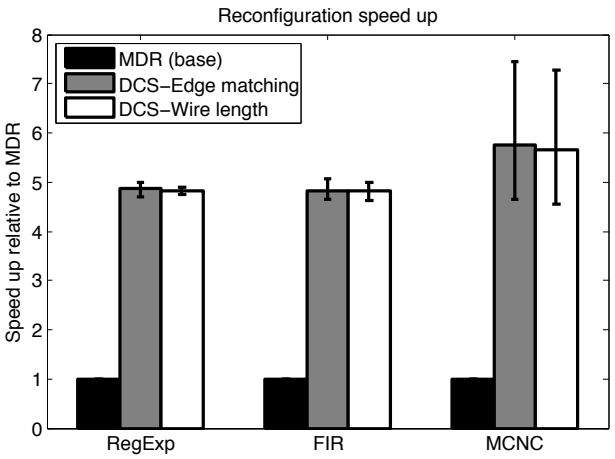

Fig. 5: Reconfiguration speed up of DCS compared to MDR.

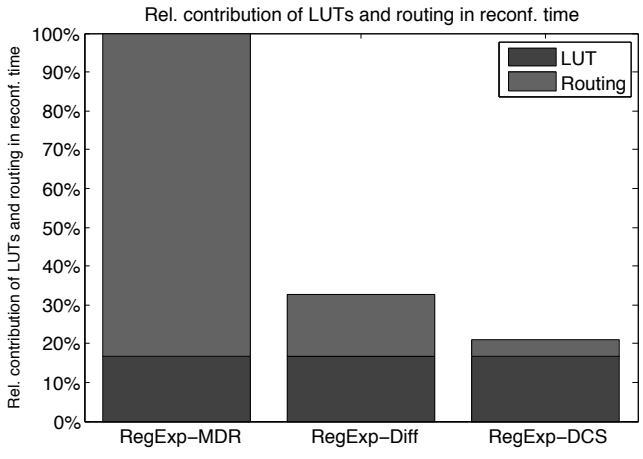

Fig. 6: Relative contribution of LUTs and routing in the reconfiguration time.

FPGA in our experiments. As recommended in [10], the square area of the FPGA and the channel width were both chosen 20% bigger than the minimum needed. This is done to allow relaxed routing.

C. Results

We point out that both our tool flow, from now on shortly referred to as DCS, and MDR have the same gains in area. For the regular expression matching application and the MCNC benchmarks, only an area of around 50% is required compared to the static implementation of the 2 modes. The adaptive filtering application requires an area which turned out to be only 33% of the generic FIR filter.

Two other metrics were used to further evaluate the quality of the multi-mode circuit: reconfiguration time and wire-length. The reconfiguration time gives an indication on how fast the system can adapt to an environmental change. Wire length is an important metric for the quality of a circuit, since it correlates with power usage and performance (maximum clock frequency) of a circuit [10]. In each experiment we compare the circuit edge matching and wire-length optimization approach to conventional MDR. We average the results over the implemented circuits and use error bars to indicate minimum and maximum values.

1) Reconfiguration time: Since the academic VPR framework is used, we could not measure an actual reconfiguration time. Instead we assume the reconfiguration time is directly proportional to the number of bits that needs to be rewritten in the configuration memory. In the case of MDR, the reconfiguration time is the time needed to write the complete reconfigurable area [1]. This comprises the bits of all the LUTs and all the bits that control the switches in the routing. As mentioned earlier, in the DCS case we also assume that all the LUTs are rewritten. We do however count only the bits in the routing that are parameterized. We assume the Boolean functions of the parameterized bits are evaluated off-line.

In Figure 5 we can see that, for typical multi-mode applications, DCS reaches a speed up of the reconfiguration process, compared to MDR, between 4.6× and 5.1×. An interesting observation is that the circuit edge matching technique and the wire length optimization technique achieve approximately the same speed up. This is explained by the fact that when connections match, the wire length also tends to decrease. Optimizing for wire length will thus also match connections, but will at the same time take wire length into consideration as we will see in the next section.

Where does this significant speed up come from? In Figure 6 we break down the reconfiguration time into the time needed for reconfiguring LUTs and routing for the regular expression matching application. We see that the number of rewritten LUT configuration cells is indeed the same, but in the case of DCS the number of routing configuration cells that is rewritten is reduced drastically compared to MDR with a factor of approximately 20. This factor consist of 2 components. The first is related to the fact that MDR is region based. Many memory cells will be rewritten with the same value, mostly zeros due to the programmability of the FPGA. To analyse this effect, we added a bar annotated with *RegExp-Diff*. This represents the case where also all LUTs are rewritten, but only the cells in the routing configuration memory that contain different bit values for the different modes are counted. We see that the first component accounts for a factor 5. The second component is due to our novel combined implementation strategy. We see the number of cells in the routing configuration memory that have a different value in the different modes is further reduced with a factor of 4 compared to MDR, thus proving the efficiency of our approach.

In current FPGAs, the reconfiguration granularity is a collection of bits called a frame. LUTs and routing memory cells reside in different frames. The next step in our research is to implement TRoute on a commercial FPGA to asses the reduction it will have in routing configuration frames that need to be reconfigured. We also plan to extend it to allocate the small number of parameterized bits in a limited amount of frames. By reconfiguring only these frames we can further

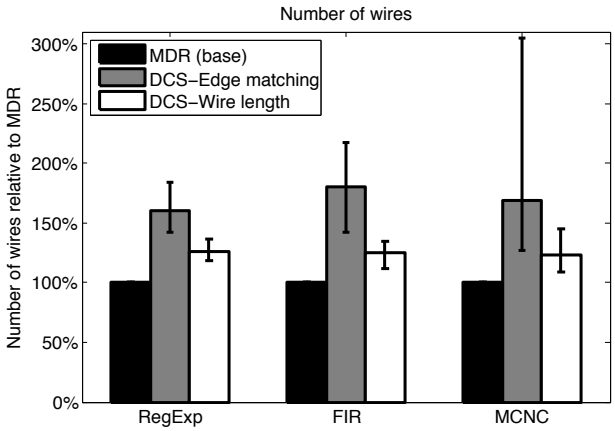

Fig. 7: Wire length of DCS compared to MDR.

reduce reconfiguration time. Given the analysis above we expect the speed up of routing reconfiguration time to be roughly between 4× and 20×.

In this paper we focused on reducing routing configuration cells that need to be rewritten. For the sake of simplicity we always assumed to write the configuration cells of all LUTs in the reconfigurable region. We point out that our results would even improve if we would count only the LUT bits that have a different value for the different modes, since this would increase the routing to LUT ratio.

2) Wire length: In our proposed tool flow the different modes are not implemented separately, as is the case in MDR, but instead a global solution is considered to generate an efficient Tunable circuit, as is explained in section III-A. In this section we assess the impact this has on the wire length. Each mode circuit uses a set of wires when it is active. We compare the size of this set in the case of implementation with MDR and DCS. This is averaged over all mode circuits.

Figure 7 shows the relative wire-length increase of an individual mode for the edge matching and the wire length optimization compared to MDR. We can clearly see that wire-length optimization significantly outperforms circuit edge matching. It seems that wire length is best optimized during the combined placement, when the Tunable circuit is formed, and not after, with TPlace, when the topology of the Tunable circuit is fixed. When the previously proposed circuit edge matching is used, the wire-length sometimes increases unacceptably with a factor of more than 2. For our novel wire-length optimization approach we see for all applications a 24% increase in wire length on average. For our target applications, regular expression matching and adaptive filtering, the minimum and maximum increase are 11% and 35%, respectively.

Note that many applications do not run at their maximum performance, because system requirements are not that stringent. Since FPGAs are mostly used for parallel applications, like regular expression matching, they rely more on massive parallelism than on high clock frequencies for performance. For many applications, the increase in wire-length is therefore not a major draw back, especially given the significant speed up of the reconfiguration process. Also the current placer and router for Tunable circuits, TPlace and TRoute, are not as mature as the conventional placer and router. As the tools evolve, we expect the results to further improve.

For the general MCNC circuits the wire-length depends more on the similarity between the circuits, which explains the higher maximum increase in wire length of 45% and higher spreading of the results.

V. Conclusion

In this paper we presented a fully automated tool flow that exploits similarities between the different modes of a multi-mode circuit and uses Dynamic Circuit Specialization to drastically reduce reconfiguration time. In our experiments we showed that for typical multi-mode applications a speed up between 4.6× and 5.1× of the reconfiguration process can be obtained compared to MDR. Of course, this doesn't come for free, the wire length of the different modes will increase slightly due to our combined implementation approach. In our experiments we showed that our novel wire length optimization approach clearly outperforms a previously proposed circuit edge matching technique. For typical multi-mode circuits, and using a wire length optimization approach, the increase in wire-length of an individual mode compared to MDR is on average 24% and between 11% and 35%.

Acknowledgment

This work was supported by the European Commission in the context of the FP7 FASTER project (#287804). Brahim Al Farisi is sponsored by IWT, Agency for Innovation through Science and Technology in Flanders.

References

[1] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *Computers and Digital Techniques*, vol. 153, no. 3, pp. 157 – 164, 2006.

[2] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," *ACM TRETS*, vol. 4, no. 4, pp. 36:1–36:24, Dec. 2011.

[3] K. Bruneel, W. Heirman, and D. Stroobandt, "Dynamic data folding with parameterizable FPGA configurations," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 4, p. 29, 2011.

[4] K. Heyse, K. Bruneel, and D. Stroobandt, "Mapping logic to reconfigurable FPGA routing," in *22nd International Conference on Field Programmable Logic and Applications, Proceedings*, 2012, pp. 315–321.

[5] E. Vansteenkiste, K. Bruneel, and D. Stroobandt, "A connection router for the dynamic reconfiguration of FPGAs," in *Lecture Notes in Computer Science*. Springer, 2012, pp. 357–364.

[6] M. Rullmann and R. Merker, "Maximum edge matching for reconfigurable computing," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 179, 2006.

[7] I. Sourdis, J. Bispo, J. Cardoso, and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *Journal of Signal Processing Systems*, vol. 51, pp. 99–121, 2008, 10.1007/s11265-007-0131-0. [Online]. Available: http://dx.doi.org/10.1007/s11265-007-0131-0

[8] Bleeding edge threats website. [Online]. Available: http://www.bleedingthreats.net

[9] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," 1991.

[10] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.