A novel tool flow for increased routing configuration similarity in multi-mode circuits

Brahim Al Farisi, Elias Vansteenkiste, Karel Bruneel 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 (RTR) of an FPGA, all the modes can be time-multiplexed on the same reconfigurable region, requiring only an area that can contain the biggest mode. Typically, conventional run-time reconfiguration techniques generate a configuration of the reconfigurable region for every mode separately. This results in configurations that are bit-wise very different. Thus, in this case, many bits need to be changed in the configuration memory to switch between modes, leading to long reconfiguration times. In this paper we present a novel tool flow that retains the placement of the conventional RTR flow, but uses TRoute, a reconfiguration-aware connection router, to implement the connections of all modes simultaneously. DRoute stimulates the sharing of routing resources between connections of different modes. This results in a significant increase in the similarity between the routing configurations of the modes. In the experimental results it is shown that the number of routing configuration bits that needs to be rewritten is reduced with a factor between 2 and 4 compared to conventional techniques.**

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, FPGA resources can be reused between circuits, making it possible to use smaller and thus cheaper FPGAs. However, at the time interval boundaries, the problem at hand will change. Then a significant period of time, called the reconfiguration time, is needed to reconfigure the FPGA.

A multi-mode circuit implements the functionality of a limited number of circuits, called mode circuits or modes. At any given time only one mode circuit needs to be realised. Also, different modes often exhibit a lot of similarity on the circuit-level, 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.

Brahim Al Farisi is sponsored by IWT, Agency for Innovation through Science and Technology in Flanders

Using run-time reconfiguration of an FPGA, all the modes of a multi-mode circuit can be time-multiplexed on the same FPGA area, requiring only an area that can contain the biggest mode. This way significant area savings can be realized compared to a static implementation of the multi-mode circuit that uses space-multiplexing to switch between modes.

In conventional RTR systems, a configuration is generated for every mode by implementing it separately in the reconfigurable region. The mode configurations are bit-wise very different and when switching between modes many bits need to be changed in the configuration memory. This leads to long reconfiguration times, making RTR less useful for more dynamic applications [8].

During the reconfiguration process most time is spent in writing the routing configuration. This is the portion in the configuration memory that controls the reconfigurable interconnection network. In this paper we focus on increasing the similarity of the routing configurations of the different modes of a multi-mode circuit. We present a novel, automated tool flow that retains the placement of conventional RTR, but uses TRoute to route the connections of all modes simultaneously. DRoute is a reconfiguration-aware connection router that stimulates sharing of routing resources between the connections of different modes, which results in an increased similarity of the mode configurations. Experimental results show a reduction between $2\times$ and $4\times$ in the number of routing configuration memory cells that needs to be rewritten compared to conventional techniques. We also research the impact this has on the wire length of the individual mode circuits as the number of modes of a multi-mode circuit increases.

Our paper starts with an overview of how TRoute works in Section II. In Section III, we compare the conventional RTR tool flow to our new, fully automated flow for multi-mode circuits. The experiments and results are discussed in Section IV. Finally, we conclude in Section V.

II. TRoute: a reconfiguration-aware connection router

A conventional router calculates the Boolean values that need to be stored in the configuration bits of the configurable interconnection network so that the physical logic blocks are connected as is specified by the nets in the mapped circuit. The main algorithm used to solve this problem is PATHFINDER [7].

PATHFINDER presents the available routing resources of the FPGA in an easy-to-explore data structure, the routing resource graph (RRG). The RRG is a directed graph, where each node represents a routing wire on the FPGA and each directed edge represents a routing switch on the FPGA.

In the PATHFINDER algorithm the connections that need to be routed are organized in nets. These are sets of connections that share the same source. Connections of a same net are allowed to share resources. In every routing iteration, the algorithm rips up and reroutes all the nets in the input circuit. These iterations are repeated until no shared resources exist between nets or, in other words, the routing trees of the nets are disjoint. This is achieved by gradually increasing the cost of sharing resources between nets, a technique called negotiated congestion. The cost function of a node in the RRG is

$$cost(n) = b(n) \cdot p(n) \cdot h(n), \tag{1}$$

where $b(n)$ is the base cost, $p(n)$ is the present congestion penalty and $h(n)$ is the historical congestion penalty. More details on PATHFINDER can be found in [2].

TRoute is the reconfiguration-aware router used in the Dynamic Circuit Specialization (DCS) tool flow presented in [3]. This DCS tool flow takes in a design with slowly varying inputs, called parameters, and generates a parameterized configuration. This is an FPGA configuration that expresses the configuration bits as a Boolean function of the parameters. Whenever the parameters change value, the Boolean functions are re-evaluated and written in the configuration memory of the FPGA.

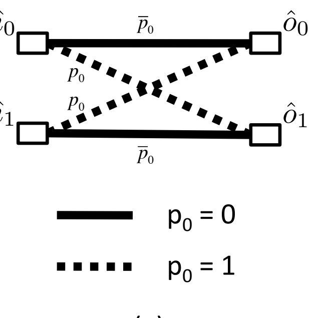
An important concept in DCS is a Tunable Connection (TCON). This is a connection, with which a Boolean function of the parameters is associated, called the activation function. The connection between the source and the sink of the TCON only needs to be realized in the FPGA fabric for parameter values for which the activation function evaluates to $True$. In contrast to nets, TCONs can legally share a node in the RRG. This is allowed when they have the same source or when they are not active at the same time. The activation functions are used to detect which TCONs are mutually exclusive in time.

TRoute is based on the PATHFINDER algorithm and is developed to route a set of Tunable connections [4]. Instead of nets, TRoute rips up and reroutes Tunable connections. The cost function of a node in the RRG, in the case of TRoute, is

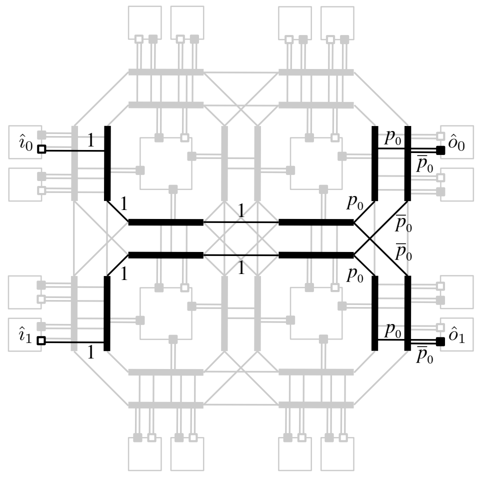
$$cost(n) = \frac{b(n) \cdot p(n) \cdot h(n)}{share(n)}, \tag{2}$$

where $b(n)$, $p(n)$, $h(n)$ are as in Equation (1) and $share(n)$ is the number of TCONs that legally share a node. Clearly, TRoute does not only stimulate the overlap between TCONs with the same source, but also between TCONs that have disjoint activation functions.

In Figure 1(a) an example is shown of a set of Tunable connections that implement the functionality of a four way switch. The straight connections are realized when $p = 0$, the crossed connections when $p = 1$. In Figure 1(b) the implementation is shown of this set of TCONs, as generated by TRoute on a 2×2 island-style FPGA. To generate the parameterized configuration bits of the switches the activation functions of the TCONs that use the switch are logically added (OR). Note that a lot of static ones are generated. This is because TRoute stimulates sharing of resources between connection with disjoint activation functions. In this example only eight configuration bits are parameterized, the rest are static. It is straight-forward to generate a specialized configuration corresponding to one of the connection patterns of the four-way switch. Only the parameterized bits need to be evaluated and written in the configuration memory.



(a)



(b)

Fig. 1: (a) Schematic representation of a set of TCONs with the functionality of a four-way switch. (b) Implementation of that set of TCONs (black) in a simple 2×2 island style FPGA resource graph (grey). Wires are solid lines; Edges are thin lines; Sources are open boxes; And sinks are filled boxes.

III. Run-time reconfiguration of multi-mode circuits

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 the implemented function, a period of time is

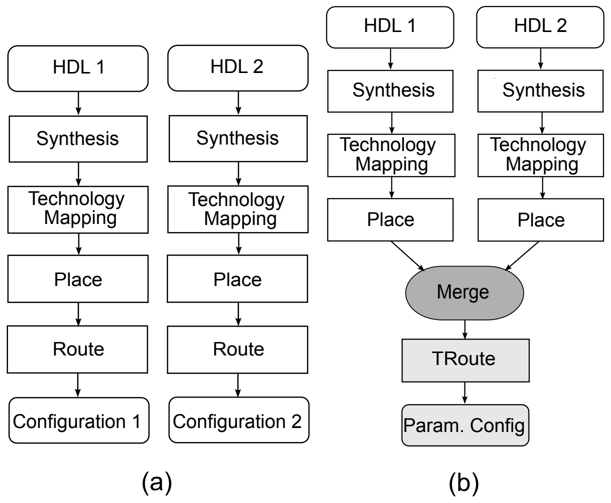


Fig. 2: The conventional RTR tool flow (a), compared to our novel approach which uses a merge and TRoute (b).

needed to rewrite the configuration memory. This is called the reconfiguration time. The subsystem that performs the reconfiguration is called the reconfiguration manager and is generally implemented in software. In this section we discuss two techniques that use run-time reconfiguration to implement multi-mode circuits: the conventional run-time reconfiguration flow and our novel approach using TRoute.

A. Conventional RTR flow

The conventional RTR tool flow implements every mode 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 2(a). For every mode a configuration is generated, that contains the binary values needed to write the configuration memory of the reconfigurable region. To switch between the different modes the reconfiguration manager writes the reconfigurable region with the appropriate configuration. In this paper we consider two ways of rewriting the configuration memory in the conventional flow: modular-based dynamic reconfiguration (MDR) and difference-based dynamic reconfiguration (DDR) [10]. In the modular based approach the configuration granularity is the complete reconfigurable region. In the difference-based approach the configuration granularity is a single bit.

B. Merging tool flow using TRoute

Our proposed tool flow is presented in Figure 2(b). Instead of running the tool flows completely separately for the different modes, the idea is to have a combined implementation of the modes at a certain point in the flow. In this case, the tool flow is run separately until placement, generating a placed design for each mode. Then the connections of all the modes are merged into one set of Tunable connections. During this merging step, the connections are automatically annotated with the appropriate activation function. The set of merged TCONs is then routed with TRoute, to generate a parameterized configuration.

Figure 3 illustrates how the merged routing using TRoute reduces the number of bits that need to be rewritten in the configuration memory compared to the conventional RTR flow. In Figure 3(a) a set of TCONs is shown that represents a simple 2:1 multiplexer we would like to implement using run-time reconfiguration. The goal is to obtain two FPGA configurations. One corresponding to connection (i_0, o_0), the other one to connection (i_1, o_0). By reconfiguring the FPGA during run-time with the appropriate configuration we can choose which input gets connected with the output.

Figures 3(b) and (c) show a possible implementation of the connections (i_0, o_0) and (i_1, o_0) respectively, on a simple 2×1 FPGA, using the conventional RTR flow. Although the wire length of each of the implementations is optimal, we note that the switches being used in the two configurations are completely different. In this example the bits of all the 10 used switches would need to be changed to obtain an appropriate configuration. In Figure 3(d) we see the same example implemented using TRoute. TRoute stimulates sharing of routing resources between connections of different modes. This results in a parameterized configuration with only 2 parameterized bits. Only these bits need to be rewritten to specialize the parameterized configuration into the appropriate regular configuration.

IV. Experiments and Results

A. Benchmarks

To validate our proposed tool flow we conducted 2 experiments that use different applications. In the first experiment a typical multi-mode application was used, namely a regular expression matching (RegExp) application. In [9] a tool was developed that can generate a hardware engine, written in VHDL, that matches a certain regular expression. We chose the regular expressions out of the Bleeding Edge rules set [1] and with this tool generated the corresponding circuits. In the second experiment the mode circuits were benchmarks chosen out of the general MCNC suite that were of similar size compared to the first experiment. For every set of mode circuits the minimum, average and maximum number of LUTs are reported in Table I.

In each experiment 10 multi-mode circuits were generated by picking a combination of M mode circuits out of the generated circuits, where M was varied between 2 and 5. This way we can analyze the quality of the implementation of the multi-mode circuits as the number of modes increases.

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

	Minimum	Average	Maximum
RegExp	224	243	261
MCNC	264	310	404

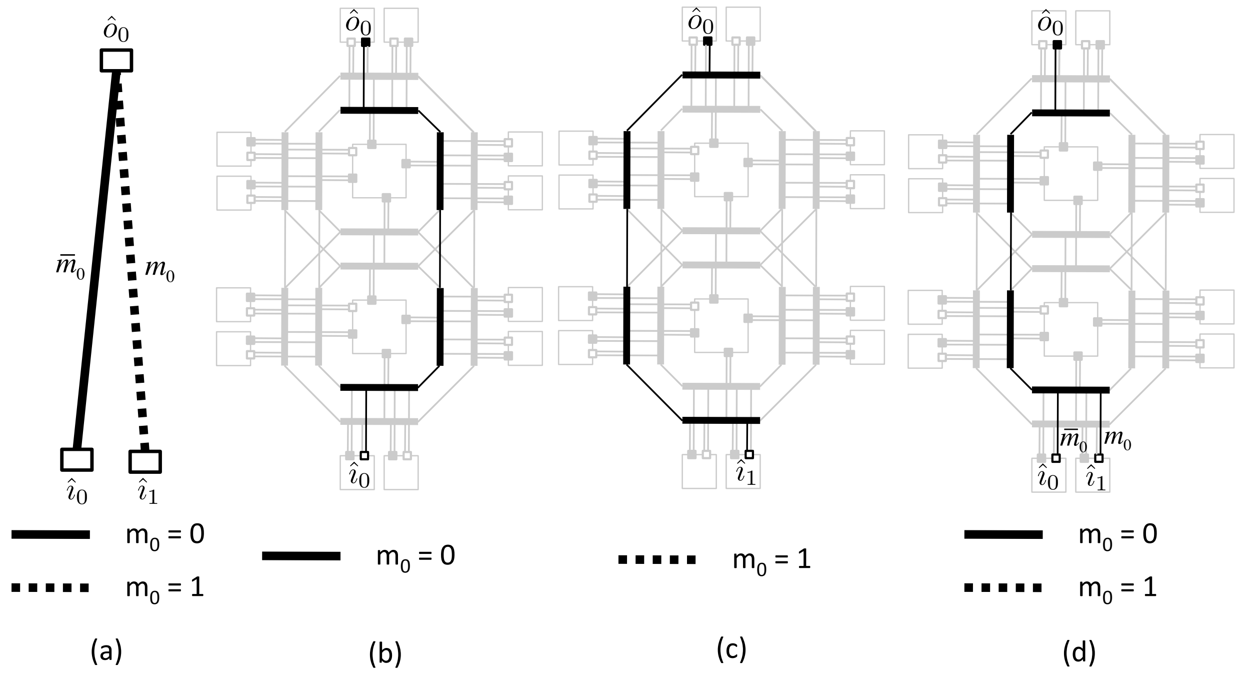


Fig. 3: Schematic representation of a set of TCONs with the functionality of a 2:1 multiplexer (a). Implementation of that set of connections using the conventional RTR flow (b)(c), compared to the parameterized configuration generated by TRoute (d).

B. FPGA architecture

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 [2]. VPR is the most commonly used academic tool for place and route algorithms in FPGAs. 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, the routing resource graph [2].

Since there is no other functionality implemented on the FPGA, the reconfigurable region comprises the complete FPGA in our experiments. As recommended in [2], 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 novel tool flow and the conventional RTR flow have the same gains in area. For the regular expression matching application and the MCNC benchmarks, only an area of around $1/M$ is required compared to the static implementation of M modes.

Two other metrics were used to further evaluate the quality of a multi-mode circuit: reconfiguration time and wire length. The reconfiguration time gives an indication on how fast the system adapts 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 [2]. In each experiment we compare our approach to difference-based dynamic reconfiguration (DDR) and modular-based dynamic reconfiguration (MDR), as the number of modes M increases. We average the results over the implemented circuits. We also use error bars to indicate minimum and maximum values compared to the average.

1) Reconfiguration time: First, in section a, we focus on the effects of using our new tool flow on the reconfiguration time of only the routing. Next, in section b, we also take a look at the effect this has on total reconfiguration time, which includes the look-up tables (LUTs).

In current FPGAs, the reconfiguration granularity is a collection of bits called a frame. LUTs and routing configuration bits reside in different frames. Since the academic VPR framework is used to implement TRoute, we could not measure a frame-based reconfiguration time. Instead we assume the reconfiguration time to be linear with the number of bits rewritten in the configuration memory. We compare our approach with both the modular and difference-based approach of reconfiguring in the conventional flow. The granularity of frame-based reconfiguration is between that of the modular and difference based approach, thus a frame-based reconfiguration time will also be between these bounds.

a) a) Routing reconfiguration time: In the case of MDR all the routing bits in the configuration memory of the reconfigurable region are rewritten. For DDR and our approach we make a distinction between static and dynamic bits. Static bits are bits in the configuration memory that have the same value for all the modes, the rest is called dynamic. In the case of DDR and our novel approach we only count the dynamic

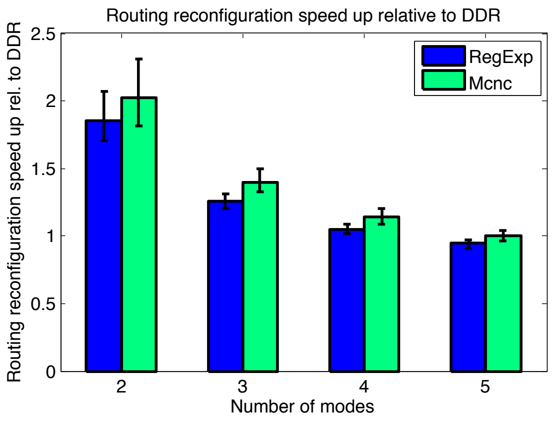


Fig. 4: Routing reconfiguration speed up rel. to DDR.

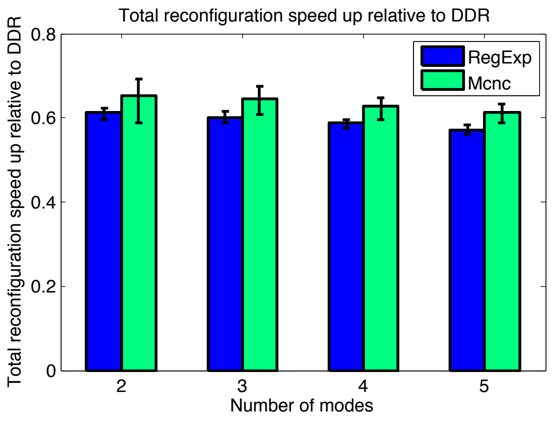


Fig. 6: Total reconfiguration speed up rel. to DDR.

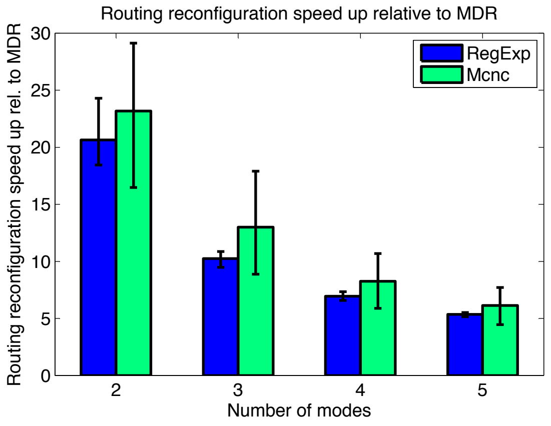


Fig. 5: Routing reconfiguration speed up rel. to MDR.

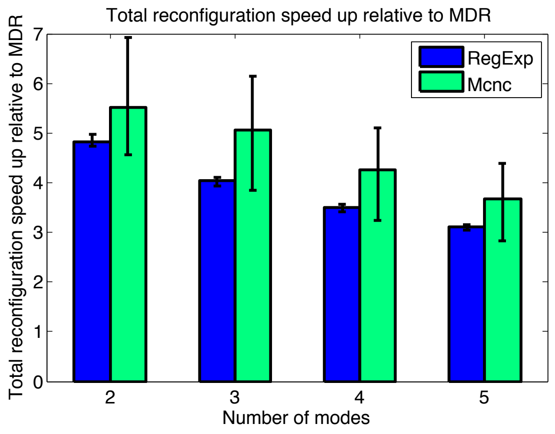


Fig. 7: Total reconfiguration speed up rel. to MDR.

bits. In Figures 4 and 5 we see the routing reconfiguration time speed up of our approach compared to DDR and MDR for the 2 benchmarks, the regular expression matchers and MCNC circuits. The results for both benchmarks are similar. Our tool flow thus works not just for typical multi-mode circuits, that have some similarity between modes. It also works for multi-mode circuits where the modes are general MCNC benchmarks.

In Figure 4 we see that our novel tool flow reduces the number of dynamic bits with a factor of 4 for 2 modes. This factor decreases gradually to 2 for 5 modes. It is clear that using our novel tool flow the similarity of the routing configurations is increased significantly. A lot less routing bits need to be altered in the configuration memory.

As expected, the speed up compared to MDR, shown in Figure 5, decreases as the number of modes increases. This is because for MDR the complete routing configuration memory is rewritten in all cases. The number of parameterized bits, however, increases with the number of modes. The ratio decreases from around 20 for 2 modes to around 5 for 5 modes.

Given the analysis above we expect the speed up of routing reconfiguration time to be roughly between 4× and 20× for 2 modes. This would gradually decrease to a speed up between 2× and 5× for 5 modes. We note that these values are measured at minimum channel width. Since FPGAs tend to be over provisioned in routing [6], [5], these are likely underestimates.

b) b) Total reconfiguration time: In this section we also assess the effect our new tool flow has on the total reconfiguration time. This comprises the routing reconfiguration time, discussed in the previous section, and the LUT reconfiguration time. For the sake of simplicity we assume to write all the configuration cells of all LUTs in the reconfigurable region, for our approach and both MDR and DDR. 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.

In Figure 7 we can see that, compared to MDR, our tool flow reaches a speed up of the total reconfiguration process of 5× for 2 modes and decreases to a speed up of 3× for 5 modes. Compared to the DDR approach the speed up, shown in Figure 6, is around 1.5 for all modes. As mentioned before, we expect the speed up of a frame-based reconfiguration approach to be between that of DDR and MDR.

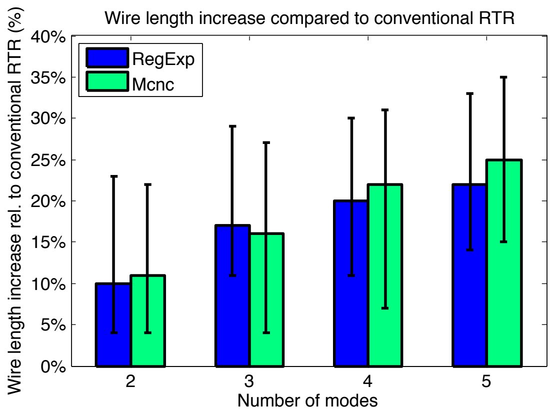


Fig. 8: Wire length of our approach relative to conventional RTR.

2) Wire length: In our proposed tool flow the different modes are not routed separately, as is the case in the conventional RTR flow, but instead a global solution is considered. 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 the conventional RTR flow and our new tool flow. We average over all mode circuits. We also use error bars to indicate minimum and maximum values compared to the average. The modular based and difference based approach have the same wire length since they only differ in the way they rewrite the configuration memory.

Figure 8 shows the relative wire length increase of our approach compared to the conventional RTR flow. On average there is, for both applications, a limited 10% increase in wire length for 2 modes. As the number of modes increases, the additional wire length increases too, but less than linearly. For 5 modes the average wire length increase is 25%. The minimum and maximum wire length increase goes from 4% and 22% for 2 modes to 15% and 33% for 5 modes, respectively.

The importance of this wire length increase is dependent on the application. Note, however, that there are applications that do not run at their maximum performance, because system requirements are not that stringent. Since FPGAs are also used for parallel applications, like regular expression matching, they often rely more on massive parallelism than on high clock frequencies for performance. There are therefore applications, for which the increase in wire length is not a major draw back, especially given the significant speed up of the reconfiguration process. Also the current version of TRoute is not as mature as the conventional router. As the tool evolves, we expect the results to further improve.

V. Conclusion

Conventional RTR implements the different modes of a multi-mode circuit completely separately. This results in routing configurations that are bit-wise very different and thus many bits need to be changed to switch between modes. In this paper we presented a fully automated tool flow that considers a combined implementation of the modes. It uses TRoute to drastically reduce the number of bits that need to be changed in the routing. TRoute routes the connections of all the modes simultaneously and stimulates sharing of routing resources between connections of different modes. In our experiments we showed that the number of bits that need to be rewritten in the routing configuration memory is reduced with a factor 4 compared to the conventional RTR flow. An attempt was made to assess the impact on total reconfiguration time: our results suggest that a speed up between 1.5 and 5 can be obtained using this technique. Of course, this does not come for free, the wire length of the different modes increases slightly due to our combined routing approach. In our experiments we showed that this increase is limited.

VI. Future work

The results in this paper show that there is already much to be gained in reduction of reconfiguration time using a merged routing approach. The next step in our research is to implement TRoute on a commercial FPGA to assess the exact 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 reduce reconfiguration time. In this work we retained the individual placements of the modes and researched a merged routing approach. We are also planning to explore optimizations during placement to further reduce reconfiguration time.

References

[1] Bleeding edge threats website.
[2] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.
[3] K. Bruneel, W. Heirman, and D. Stroobandt. Dynamic data folding with parameterizable FPGA configurations. *ACM Transactions on Design Automation of Electronic Systems*, 16(4):29, 2011.
[4] K. Bruneel and D. Stroobandt. Troute: a reconfigurability-aware fpga router. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 207–218, 2010.
[5] J. Lee. *An analytical model describing the performance of application-specific networks-on-chip on field-programmable gate arrays*. PhD thesis, 2010.
[6] Y. Lu, J. Mccanny, and S. Sezer. The impact of global routing on the performance of nocs in fpgas. *Reconfigurable Computing and FPGAs, International Conference on*, 0:369–374, 2011.
[7] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *FPGA*, pages 111–117, 1995.
[8] K. Papadimitriou, A. Dollas, and S. Hauck. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):36:1–36:24, 2011.
[9] I. Sourdis, J. Bispo, J. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51:99–121, 2008.
[10] Xilinx. *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*.