A Connection Router for the Dynamic Reconfiguration of FPGAs

Elias Vansteenkiste, Karel Bruneel, and Dirk Stroobandt

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

Abstract. Dynamic Circuit Specialization (DCS) is a new FPGA CAD tool flow that uses Run-Time Reconfiguration to automatically specialize the FPGA configuration for a whole range of specific data values. DCS implementations are a factor 5 faster and need a factor 8 less LUTs compared to conventional implementations. We propose a novel routing algorithm for reconfigurable routing, called the Connection router. In contrast to TROUTE, another reconfiguration-aware router, our new router is fully automated and far more scalable.

Keywords: FPGA, Run-Time Reconfiguration, PATHFINDER, Dynamic Circuit Specialization (DCS) , CAD tool flow, Reconfigurable routing

1 Introduction

Run-time reconfiguration (RTR) enables more efficient utilization of Field Programmable Gate Arrays (FPGA) by specializing an FPGA's functionality for the current problem specifics. This can be done by simply writing a specialized configuration in the FPGA's configuration memory. A specialized configuration uses fewer resources and can attain faster clock speeds than a generic implementation. The downside is the specialization overhead – the time needed to generate a specialized configuration and write it in the configuration memory. Generating a specialized configuration with a conventional FPGA tool flow can take in the order of minutes to hours, which is unacceptable for most applications.

In [3] the TLUT method has been developed. It is able to produce specialized configurations several orders of magnitude faster than a conventional FPGA tool flow, without sacrificing the quality (speed and area) of the specialized configurations. The method produces a specialized configuration in two steps. Off line, a parameterized configuration is created. This is a closed-form multi-valued Boolean function that expresses the FPGA configuration as a function of a set of parameter values. Online, this Boolean function is evaluated every time the parameter values change, in order to produce a specialized configuration. Since evaluating a closed-form Boolean function is a lot faster than running a conventional FPGA tool flow, the TLUT method greatly reduces the specialization overhead. E.g., the TLUT method can produce specialized FIR configurations (8-bit input, 8-bit coefficients and 128 taps) in only 1.3 ms, while the conventional method needs 35,634 ms.

Parameterized configurations produced by the TLUT method only express the truth tables of LUTs as a function of the parameters. All routing between LUTs is fixed. This leads to good quality specialized configurations. However, it has been shown [4] that also expressing the routing bits as a function of the parameters (TCON method) leads to specialized configurations with an even better quality. The TCON method produces parameterized configurations starting from an RT level HDL description, using adapted versions of synthesis, technology mapping, placement and routing. Our research group is working on adapted algorithms for each of these steps, but in this paper we focus on the routing step.

In [4] a first reconfigurability-aware router, called TROUTE, is presented. Although TROUTE produces good quality results in a reasonable time, manual optimization of the input is needed. The new routing algorithm presented in this paper, solves this problem. It produces good quality results in a reasonable time, independent of the way the input problem is represented.

In the background section we describe the TROUTE algorithm and its limitations. The Connection router is presented in section 3 and the performance of the Connection router is discussed in section 4.

2 Background

2.1 Tunable circuit

As explained in [4], the technology mapping step of the TCON method produces a *tunable circuit*. This is a circuit containing two types of functional blocks, Tunable LUTs (TLUT) and Tunable Connections (TCON). A TLUT is a LUT with the truth table bits expressed in terms of parameters. A TCON is a functional block, which has any number of input ports (set $\mathcal{I}$)) and any number of output ports (set $\mathcal{O}$). Every TCON has a connection function $\zeta_p : P' \rightarrow (\mathcal{O} \rightarrow \mathcal{I})$ that expresses how the output ports are connected to the input ports given a parameter value $p \in P'$. P' is the subset of $\{0,1\}^N$ that contains all possible parameter values. A TCON is an abstraction of reconfigurable interconnect between the LUTs.

In what follows we will use a TCON with the functionality of a 2×2 crossbar switch as example. This TCON has two inputs $\mathcal{I} = \{\hat{i}_0, \hat{i}_1\}$ and two outputs $\mathcal{O} = \{\hat{o}_0, \hat{o}_1\}$. A schematic of this TCON can be seen in Figure 1.

2.2 The TCON routing problem and TROUTE

The TCON routing problem is defined as follows. Given a tunable circuit containing TLUTs and TCONs and a physical location for each of the TLUTs, express the FPGA's routing bits as a Boolean function of the parameter inputs, so that the connections represented by the TCONs in the tunable circuit are realized for every possible parameter value in P'.

In [4] the TCON routing problem is solved by a reconfiguration-aware router, called TROUTE. TROUTE is based on the PATHFINDER algorithm. The available routing resources of the FPGA are represented in the routing resource

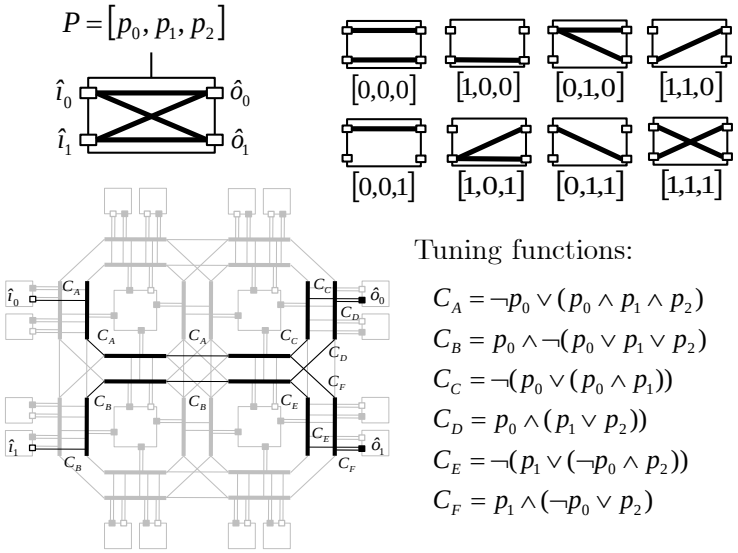

Fig. 1. TCON representation of a 2×2 crossbar switch (upper left) and the patterns and parameter values for which the patterns need to be activated (upper right). A possible implementation on a 2×2 FPGA after place and route (lower left) shows the TCON as a set of routing resources (wires and switches), where switches are controlled by Boolean functions of the parameter inputs, called tuning functions (lower right).

graph (RRG), a directed graph where each node represents a routing wire on the FPGA and each arc represents a routing switch on the FPGA. In the same way as PATHFINDER tries to find disjoint subgraphs of the RRG for each of the nets in the input circuit, TROUTE tries to find disjoint subgraphs for each of the TCONs in the tunable circuit. Both algorithms do this iteratively by ripping up and rerouting nets or TCONs, respectively. The subgraphs are calculated independently of each other using a net or a TCON router, which tries to find a minimum cost subgraph[1] but does not explicitly force the subgraphs to be disjoint. In order to make the subgraphs disjoint, the cost of the routing resources is manipulated by a mechanism called negotiated congestion (see Figure 2). For more information on negotiated congestion we refer to [2, 7].

In order to find a minimum cost subgraph that implements a TCON, TROUTE looks at a TCON as a set of connection patterns. The connection patterns of the 2×2 crossbar switch example are depicted in Figure 1 (upper right). TROUTE routes a TCON by routing each pattern separately. Nets that are part of the same routing pattern are active at the same time and thus have to be disjoint (in order to avoid short circuits). However, two nets that are part of different patterns are never activated at the same time and can thus share routing resources. This

[1] The subgraph cost equals the sum of the costs for all routing resources it contains.

```
while (congestedResourcesExist()):
 for each TCON τ do:
  τ.ripUpRouting()
  routeTCON(τ)
  τ.resources().updateSharingCost()
 allResources().updateHistoryCost()
```

Fig. 2. Pseudo code for the negotiated congestion loop of the TCON router.

```
while (congestedResourcesExist()):
 for each connection ζ do:
  ζ.ripUpRouting()
  ζ.path = dijkstra(ζ.source, ζ.sink)
  ζ.resources().updateCongestionCost()
 allResources().updateHistoryCost()
```

Fig. 3. Pseudo code for the negotiated congestion loop of the Connection router.

last property is used to minimize the routing cost of a TCON by maximizing the overlap among different patterns. The algorithm loops over all patterns of the TCON and stimulates overlap between patterns by setting the cost of previously used resources (for other patterns within the same TCON) to 0. Within a pattern, the algorithm loops over all nets in the current pattern and routes them using the net router. In order to keep the nets disjoint, the cost of previously used resources (for nets within the same pattern) is set to ∞. The negotiated congestion cost of a node is given by $c(n) = b(n) \cdot h(n) \cdot p(n)$, where $b(n)$, $h(n)$, and $p(n)$ are the base, history and present congestion cost.

2.3 Limitations of TROUTE

The TCON router avoids overlap between nets in the same pattern by setting the cost of the routing resources used by the already routed nets to ∞. This mechanism is called obstacle avoidance. It is well known that obstacle avoidance fails to find good quality solutions for complex circuits [7], due to the *enclosure problem*. The terminals of a net get enclosed by resources that are already used by other nets in the same pattern. Thus, when patterns become too complex the TCON router fails to find a routing graph. This is the first limitation. The second limitation is that the run time of the TCON router scales exponentially with the complexity of the TCON. In worst case scenario, the number of times that the Dijkstra algorithm needs to be invoked to (re)route a TCON with inputs $\mathcal{I}$ and outputs $\mathcal{O}$, is equal to $|\mathcal{I}||\mathcal{O}|(|\mathcal{I}|+1)^{|\mathcal{O}|-1}$, which is clearly exponential in the number of outputs. The TCON of the 2×2 crossbar in Figure 1 has 8 patterns and needs 12 Dijkstra invocations. For a 4×4 crossbar the number of patterns increases to 624 patterns and the number of invocations to 2000.

In [4] these two issues were addressed by manually reducing the complexity of the input tunable circuit. This was done by splitting up larger TCONs into several smaller TCONs. E.g., a TCON representing a 4×4 crossbar was split into four 4:1 multiplexers. The patterns of these TCONs contain only one net, and thus the obstacle avoidance problem was avoided. Additionally, each TCON contains only four patterns, which greatly reduces the routing complexity. In the next section, we propose a new algorithm that is not dependent on the way the input tunable circuit is represented and thus solves the problems associated with TROUTE.

3 The Connection router

In order to solve the problems of TROUTE, we developed a new routing algorithm, the Connection router. The Connection router uses a connection-based representation instead of the TCON-based representation for the reconfigurable routing in a tunable circuit. The connections in this representation are again associated to a connection condition, expressed in terms of the parameter inputs of the design. The condition is true for those parameter values that require the connection to be activated. The connection representation of the 2×2 crossbar switch example in Figure 1 is given by C_C for connection (i_0, o_0), C_F for (i_0, o_1), C_D for (i_1, o_0) and C_E for (i_1, o_1).

The pseudo code of the Connection router can be found in Figure 3. The negotiation loop of the Connection router will now rip up and reroute connections. Dijkstra's algorithm is used to calculate the lowest cost path between the source and the sink of the connections.

Connections are allowed to share resources, in contrast to nets/TCONs in PATHFINDER/TROUTE. Indeed, connections are allowed to overlap if they carry the same signal or if they are not active at the same time. Note that this complicates things for the negotiated congestion mechanism. To update the history $h(n)$ and the present congestion cost $p(n)$, the negotiated congestion mechanism needs to know how congested a routing node is. PATHFINDER/TROUTE simply counts the number of nets/TCONs that were sharing the node, but now things are more complicated. The Connection router needs to find a minimum partition of the connections that share the routing resource under consideration, so that each partition only contains connections that are allowed to overlap. This reduces to a so called minimum clique cover problem, which is NP-complete [6]. This problem needs to be solved in the inner loop of the routing algorithm and may lead to exuberant run-times. Therefore, the Connection router approximates the congestion and the legal sharing by only allowing overlap between connections that either share the same source or the same sink. Connections that share the same source carry the same signal and thus are allowed to overlap. Connections that share the same sink are allowed to overlap because they are never active at the same time, at least if we assume that the input to the router is a legal tunable circuit (no shorts). These simplified overlap rules allow a simplification of the minimum clique cover problem.

To compare the different overlap possibilities the connection router needs to accurately assess how a node used in a connection contributes to the cost of the complete solution. If a node is legally shared between several connections the cost should be divided, so that the sum of all the connection costs equals the cost of the complete solution. The cost of a node is thus given by:

$$c(n) = \frac{b(n) \cdot h(n) \cdot p(n)}{share(n)}, \tag{1}$$

where $share(n)$ is the number of connections that legally share the node n. Once the partitioning from the previous paragraph is performed, this number is equal to the cardinality of the partition the current connection is part of.

The algorithm described in this section solves both problems described in Section 2.3. The enclosure problem is solved because obstacle avoidance is not used anymore. The scalability problem is also greatly improved. To show this, we again calculate the worst-case number of Dijkstra invocations that are needed to route a TCON. A TCON with inputs $\mathcal{I}$ and outputs $\mathcal{O}$ will in worst case lead to $|\mathcal{I}||\mathcal{O}|$ different connections. This is also the number of Dijkstra invocations needed to route the TCON. To route a 4 by 4 crossbar, only 16 Dijkstra invocations are needed, compared to the 2000 Dijkstra invocations needed to route the crossbar using TROUTE.

4 Experiments and results

To validate and compare the reconfiguration-aware routers, we used Multistage Interconnect Networks that are known as Clos Networks [5]. The connection router can handle every routing architecture that can be represented by an RRG.[2] Here we used a simple FPGA architecture[3] with logic blocks containing one 4-LUT and one flip-flop. The wire segments in the interconnection network only span one logic block. The architecture is specified by three parameters: the number of logic element columns ($cols$), the number of logic element rows ($rows$) and the number of wires in a routing channel (W).

Our Clos network uses 4×4 crossbar switches as building blocks. The TCON representation of one such switch' functionality needs 624 patterns. We can reduce this number to 16 by dividing the functionality over 4 TCONs, as described in section 2 (resulting in 4 patterns per TCON). The process of finding the perfect division of the functionality over different TCONs, such that the size of the TCON is acceptable and the patterns only contain one net, is difficult to automate. Nothing can be said about the existence of such representation. The Connection router avoids manual optimization, greatly reducing the design effort.

The goal of this experiment is to look at the possibilities of the Connection router and how they approach those of TROUTE, applied on a manually optimized representation of the tunable circuit. We compare three sizes, 16×16 (3 stages), 64×64 (5 stages) and 256×256 (7 stages). In our implementation the crossbar switches in the even stages are implemented using TLUTs and in the odd stages using reconfigurable routing. This results in a good balance between TLUTs and reconfigurable routing. We first tried to route the unmodified representation with TROUTE. These trials failed due to the previously mentioned (subsection 2.3) enclosure problem. Secondly we manually optimized the representation, as described in the previous paragraph, and routed again with TROUTE (these results are denoted *Troute**). In the last approach (*Con*) we used the Connection router to route the unmodified representation. The placement is done using an adapted version of the VPR routability-driven placer, called TPLACE (beyond the scope of this paper).

[2] For more information about RRG generation, see [1].

[3] A description of this architecture is provided with the VPR tool suite in `4lut_sanitized.arch`.

Table 1. Properties of six multi stage Clos network implementations. The numbers between parentheses are relative compared to the Troute* result.

Size	Type	Area			Architecture			
		LUTs	Wires		W_m	*Cols*	*Rows*	W
16	*Troute**	16	456	(1.00)	6	8	8	7
	Con	16	428	(0.94)	5	8	8	7
64	*Troute**	128	4178	(1.00)	11	18	18	13
	Con	128	4372	(1.04)	11	18	18	13
256	*Troute**	768	24851	(1.00)	19	39	39	24
	Con	768	35229	(1.42)	20	39	39	24

For each size we measured the number of wires and the minimum channel width (W_m). Table 1 shows the results. The parameters of the FPGA architecture and the number of LUTs for each size are also shown. As suggested in [1], we ensure low-stress place and route by choosing the number of LUTs in the FPGA architecture 20% larger than the number of LUTs in the circuit and the number of wires per channel 20% larger than W_m, the minimum channel width. We set the maximal routing iterations to 250 for both algorithms.

Con needs up to a factor 1.4 more wires than *Troute**. The Connection router has more freedom to share or not share resources, which leads to a larger search space. The chance to find a solution with more wires is higher. The Connection router is routability-driven. It stops when a solution is found for the given track width, there is no extra wire length optimization.

The routers are routability-driven, so in Figure 4 we compare the time needed to reach a solution for a given track width. The timing experiments are done using an Intel Core 2 processor running at 2.13 GHz with 2 GiB of memory running the Java HotSpot^{TM} 64-Bit Server VM. The run-time of the Connection router is higher, due to the larger search space. This effect becomes more pronounced in case of the 256x256 network. Keep in mind that the Connection router is an automated process (no intervention of the designer is necessary), in contrast to TROUTE. The extra design time needed when using TROUTE is not accounted for in the charts. We assume the time to optimize the tunable circuit is higher than the run-time of the Connection router.

For the 16×16 network, the Connection router finds a solution with a smaller track width than TROUTE. This demonstrates that solutions with lower track widths are present in the solution space of the Connection router. For the 64×64 network, the Connection router and TROUTE have the same minimal track width. For the 256×256 network, the connection router does not find a solution for the minimal track width that TROUTE reaches, due to the larger search space. However, without the manual optimization, TROUTE would not even find a solution at all.

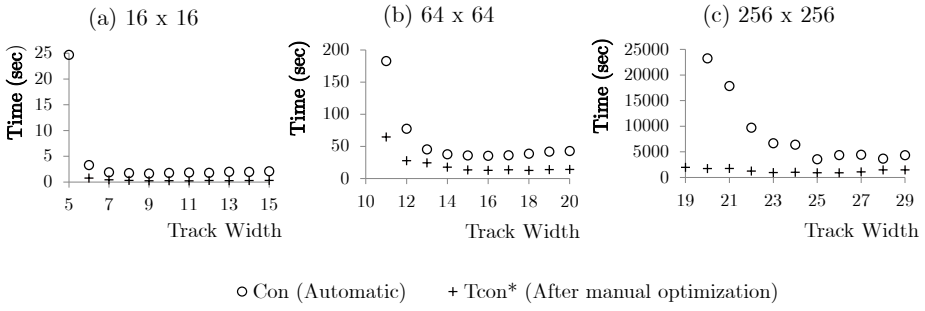

Fig. 4. Run-time to route Clos networks for a given track width. The connection router starts from a non-optimized and Troute from a manually optimized representation.

5 Conclusion and Future Work

In this paper we introduce a new reconfiguration-aware router, the Connection router. This algorithm is far more scalable than TROUTE, described in [4]. Starting from a non-optimized representation of a tunable circuit, the Connection router succeeds in routing representations that are unroutable for TROUTE. The quality of the routings produced by the Connection router approaches that of TROUTE, applied on a manual optimized representation. To achieve this, we use a connection-based representation for the reconfigurable routing.

There are many possibilities to improve the Connection router. We demonstrated that the Connection router can find solutions with lower track widths, but needs more time to find these solutions. In future research we will focus on improving the Connection router's efficiency and develop a wire length-driven and a timing-driven Connection router.

References

1. Betz, V., Rose, J., Marquardt, A.: Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, Norwell (1999)
2. Betz, V., Rose, J., Marquardt, A.: VPR: A New Packing, Placement and Routing Tool for FPGA Research. In: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, pp. 213-222. (1997)
3. Bruneel, K., Abouella, F., Stroobandt, D.: TMAP: A Reconfigurability-aware FPGA Technology Mapper. In: Design, Automation and Test Europe (2009)
4. Bruneel, K., Stroobandt, D.: TROUTE: A Reconfigurability-aware FPGA Router. In: Lecture Notes In Computer Science, vol. 5992, pp. 207-218. (2010)
5. Clos, C.: A Study of Non-blocking Switching Networks. In: The Bell System Technical Journal, XXXII, pp. 406-424. (1953)
6. Karp, K.: Reducibility Among Combinatorial Problems. In: Complexity of Computer Computations , pp. 85-103, Plenum Press. (1972)
7. McMurchie, L., Ebeling, C.: PATHFINDER: A Negotiation-based Performance-driven Router for FPGAs. In: FPGA '95, pp. 111-117. (1995)