MAXIMIZING THE REUSE OF ROUTING RESOURCES IN A RECONFIGURATION-AWARE CONNECTION ROUTER

Elias Vansteenkiste, Karel Bruneel and Dirk Stroobandt*

Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
{Elias.Vansteenkiste, Karel.Bruneel, Dirk.Stroobandt}@elis.ugent.be

ABSTRACT

Parameterised configurations for FPGAs are configuration bitstreams of which some of the bits are defined as Boolean functions of parameters. By evaluating these Boolean functions using different parameter values, it is possible to quickly and efficiently derive specialised configuration bitstreams with different properties. Generating and using parameterized configurations requires a new tool flow. In this paper we propose a novel algorithm for the routing step of this tool flow. This new router, called the connection bundle router, is able to route a circuit with parameterized interconnections. It produces routing solutions in less time (up to a factor 5,2) and with a better quality in terms of number of wires (up to 38%) and minimum track width (up to 25%) than its pre decessors. The connection bundle router is fully automated and uses a scalable connection-based representation for the parameterized interconnections in a tunable circuit.

1. INTRODUCTION

Run-time reconfiguration (RTR) allows an improved utilisation of Field Programmable Gate Arrays (FPGA). RTR enables specializing the FPGA through reconfiguring the FPGA for the current application. This can be done by writing a specialized configuration to the FPGA's configuration memory. A specialized configuration uses fewer resources and can attain faster clock speeds than a generic implementation. The downside is that the gain in efficiency can be nullified by the specialization overhead – the time needed to generate a specialized configuration and write it to 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 is proposed. It produces 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 selects the input signals that change their value much less frequent than other inputs and marks them as parameters. Then a specialized configuration is produced in two steps. Offline, a parameterised configuration is created. Parameterised FPGA configurations [2] are configuration bitstreams of which some of the bits are defined as Boolean functions of parameters. Online, these Boolean functions can be evaluated using different parameter values, so it is possible to derive specialised configuration bitstreams with different properties and/or functionality from a single parameterised configuration.

The advantage of generating specialised configurations from a parameterised configuration, instead of directly running the conventional FPGA tool flow, is the much lower time cost per specialized configuration. Generating a specialized configuration from a parameterized configuration requires only the evaluation of some Boolean functions, instead of solving the computationally hard problems contained in the full conventional tool flow, such as placement and routing.

Parameterised configurations can be used in many applications. A first example is the "hard coding" of device specific information in a production process (but with the flexibility to change the information later): e.g. a MAC address, an encryption key or calibration data. A second example is speeding up development by allowing the developer to tune coefficients, e.g. the coefficients of a DSP filter or the address range of a bus peripheral, without having to rerun the complete tool flow. Another important application of parameterised configurations is the efficient generation of configuration bitstreams for Dynamic Circuit Specialisation (DCS). With DCS, RTR of (parts of) an FPGA is used to dynamically optimise the circuit for the current situation in order to improve its performance, increase the functional density of the FPGA and consequently reduce the cost of the design [10].

Using the conventional tool flow to create the specialised

*Elias Vansteenkiste is sponsored by IWT, Agency for Innovation through Science and Technology in Flanders

configurations at run-time for DCS would in many cases take too much time (minutes to hours), while computing all specialised configurations in advance and storing them in a database quickly becomes infeasible because of the immense number of possible configurations. E.g. 2^{32} or ca. 4 billion configurations are needed for a 32-bit multiplier with a semi-constant coefficient. With a parameterised configuration on the other hand, only one configuration needs to be stored while the different specialised configurations can be generated in milliseconds [2]. For example, the TLUT method can produce specialized FIR configurations (8-bit input, 8-bit coefficients and 32 taps) in only 1.3 ms on a PowerPC 405 (PPC405) clocked at 300 Mhz (available in the Xilinx Virtex-II pro FPGAs), while the conventional method needs several minutes to produce a specialized configuration on a standard desktop PC.

The parameterized configurations produced by the TLUT method only express the truth tables of the LUTs as a function of the parameters. All the routing between the LUTs is fixed. This leads to good quality specialized configurations. However, it has been shown in [4] that also expressing the routing configuration as a function of the parameters (TCON method) leads to specialized configurations with an even better quality. As can be seen in Section 5, a 256×256 Clos switch requires 1792 LUTs when the TLUT method is used while only 768 LUTs are needed when the TCON method is used.

The TCON method produces parameterised configurations starting from an RT level HDL description, using adapted versions of synthesis, technology mapping, placement and routing. Allowing reconfiguration of routing requires changes in every step of the tool chain. At this point, our research group is working on adapted algorithms for each of the steps. An overview of the TCON method tool flow is given in Section 2.1. In this paper we focus on the routing step.

In [4] a proof-of-concept reconfigurability-aware router, called TROUTE, is proposed. Although TROUTE produces good quality results in a reasonable time, these results are heavily dependent on the way the routing problem is presented. The input is represented by a pattern-based representation. For good results manual optimization of the input is needed. More details can be fount in subsection 2.3 and 2.4. In [9] the connection router is proposed, a first approach to circumvent the problems of TROUTE. The downside of the connection router is that it takes a factor of 6 more time to generate parameterised configurations than TROUTE and the solutions contain up to 29% more wires. In this paper we present the Connection Bundle router. This new reconfiguration-aware routing algorithm produces better quality results in less time than the connection router. No manual optimization of the input is needed.

In section 2 the TCON toolflow, the TROUTE algorithm and its limitations are discussed. The scalable connection-based representation and the Connection router are described in section 3. In section 4 a mechanism to extend resource sharing and the adaptation to the routing algorithm is presented. In section 5 the performance of the routers are compared.

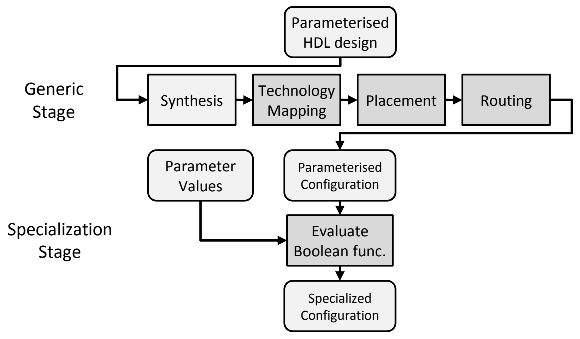


Fig. 1: TCON Tool flow for parameterised configurations

2. BACKGROUND

2.1. TCON Tool Flow

The TCON tool flow, depicted in Figure 1, consists of two stages. The generic stage is performed off-line at design time to create a parameterised configuration from an HDL description in which a number of inputs are denoted as parameters. The specialisation stage is performed every time a specialised configuration is needed to (re-)configure an FPGA. During the specialisation stage, the Boolean functions contained in the parameterised configuration are evaluated to create a fully defined configuration bitstream. This can be done efficiently using a microprocessor.

The generic stage of our new flow consists of similar, but adapted, parts as the conventional tool flow. We give a description of the changes made in each part.

2.1.1. Synthesis

The synthesis step converts a HDL description in which some inputs are annotated as parameters into a parameterised Boolean network. No significant changes need to be made to this step because parameters can be synthesised just like regular inputs.

2.1.2. Technology Mapping

During technology mapping, the parameterised Boolean network generated by the synthesis step is mapped onto the resource primitives available in the target FPGA architecture while trying to optimise the delay and area of the circuit. To generate a parameterised configuration, we want to map to primitives that can be parameterised:

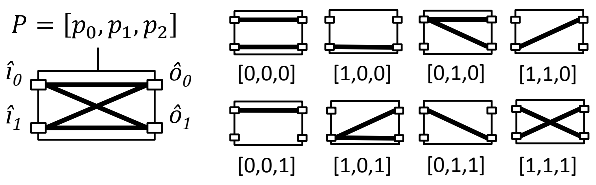


Fig. 2: A schematic of the TCON functionality of a 2×2 crossbar switch and the pattern representation. Under each pattern, the corresponding parameter values for which the pattern needs to be activated, are shown.

- Tuneable LUT (TLUT): A TLUT is a LookUp Table with the truth table expressed in terms of parameters.
- Tuneable Connection (TCON): A TCON is a parameterized interconnection between the (T)LUTs.

A TCON has any number of input ports $\mathcal{I} = \{\hat{\imath}_0, \hat{\imath}_1, \ldots, \hat{\imath}_{L-1}\}$ and any number of output ports $\mathcal{O} = \{\hat{o}_0, \hat{o}_1, \ldots, \hat{o}_{M-1}\}$. Every TCON has a connection function $\zeta_p : P \to (\mathcal{O} \to \mathcal{I})$ that expresses how the output ports are connected to the input ports given a parameter value $p \in P$. In this paper a TCON with the functionality of a 2×2 crossbar switch is used as an example. This TCON has two inputs $\{\hat{\imath}_0, \hat{\imath}_1\}$ and two outputs $\{\hat{o}_0, \hat{o}_1\}$. A schematic can be seen in Figure 2.

TCONMAP [6] is a technology mapping algorithm, able to exploit both the reconfigurable properties of the LUTs and the interconnect network of the FPGA. TCONMAP produces a tunable circuit. This is a circuit containing four types of functional blocks: (T)LUTs and (T)CONs.

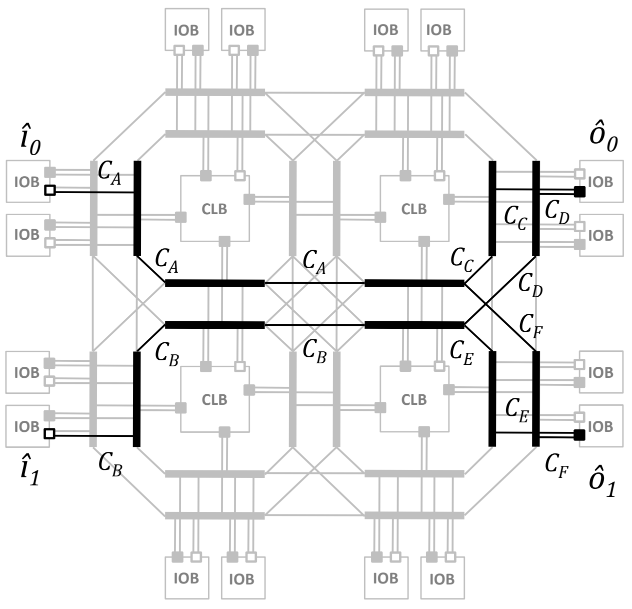
2.1.3. Placement & Routing

During the placement step a physical LUT on the FPGA is chosen to implement every instance of the (T)LUT primitives, while during the subsequent routing step routing resources are chosen to implement the (T)CONs.

The placement step makes extensive use of wirelength estimates of the connections between LUTs to optimise the routability and interconnect delay of the design. Since the TCON is very different from a static connection, new ways to estimate the length of connections are required. In what follows the routing step will be discussed in more detail.

2.2. The TCON Routing Problem

The TCON routing problem is defined as follows. Given a tunable circuit (TCONMAP) and a physical location for each of the (T)LUTs (TPLACE), express the FPGA's routing configuration as a Boolean function of the parameter inputs, so that the connections represented by the TCONs are realized for every possible parameter value in P.



Tuning functions:

$$C_A = \neg p_0 \vee (p_0 \wedge p_1 \wedge p_2)$$
$$C_B = p_0 \vee \neg(p_0 \vee p_1 \vee p_2)$$
$$C_C = \neg(p_0 \vee (p_1 \wedge p_2))$$
$$C_D = p_0 \wedge (p_1 \vee p_2)$$
$$C_E = \neg(p_1 \vee (\neg p_0 \wedge p_2))$$
$$C_F = p_1 \wedge (\neg p_0 \vee p_2)$$

Fig. 3: An implementation of the 2×2 crossbar switch example on a FPGA with 2×2 CLBs. The thick lines in the schematic represent the wires of the FPGA, the thin lines the switches

After refinement by place and route, the TCON should be realized by a set of routing resources (wires and switches), where each of the switches is controlled by a Boolean function of the parameter inputs, called a tuning function. A possible implementation of the 2×2 crossbar switch example is shown in Figure 3. The reader can verify that for every parameter value, the correct pattern is realized.

2.3. TROUTE

In all of the routers discussed in this paper, the available routing resources of the FPGA are represented 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 connection that can be made between two wires by closing a routing switch. A whole range of FPGA routing architectures can be represented by the RRG, this makes our algorithm more generic. For more information we refer to

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

Fig. 4: Pseudo code for the negotiated congestion loop of the TCON router.

```
function routeTCON(Tcon $\tau$):
  for each pattern $\pi$ in $\tau$ do:
    for each net $\eta$ in $\pi$ do:
      routeNet($\eta$)
      $\eta$.resources.setCost($\infty$)
    $\tau$.resources.setCost(0)
  $\tau$.resources.resetCost()
```

Fig. 5: Pseudo code for the TCON router function.

[1].

In [4] the TCON routing problem is solved by a reconfiguration-aware router, called TROUTE. Figure 4 shows the pseudo code for TROUTE. TROUTE is based on PATHFINDER [8]. Both algorithms try to find disjoint subgraphs of the RRG for each of the nets or TCONs in the input circuit, respectively. 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. Congestion (overuse) is allowed in the first iteration and solved in subsequent iterations by gradually increasing the cost of congested routing resources. After every routing iteration the cost of the overused nodes goes up by increasing the history congestion factor $h(n)$ of the congested nodes. During each routing iteration congestion is avoided by increasing the present congestion factor $p(n)$. The total cost of a node in the RRG during the PATHFINDER algorithm is given by $c(n) = b(n){\cdot}h(n){\cdot}p(n)$, with $b(n)$ the base cost of the node. For more information on negotiated congestion we refer to [8, 1] .

In order to find a minimum cost subgraph that implements a TCON a routing heuristic is used, the pseudo-code can be found in Figure 5. A TCON is represented as a set of connection patterns. The connection patterns of the 2×2 crossbar switch are depicted in Figure 2. A connection pattern, $\pi \in \Pi$, is one way of connecting the inputs of a TCON to its outputs, where Π is the set of all possible connection patterns. Mathematically a connection pattern is a function, $\pi(\hat{o}) : \mathcal{O} \rightarrow \mathcal{I}$, that maps every output of the TCON to at most one of its inputs. The connection patterns themselves can be represented as a set of nets. Each net connects one of the TCON's inputs to one or more of its outputs.

Each pattern is routed separately. The union of the routing graphs of all the patterns is the routing graph of the TCON. Nets that are part of the same routing pattern need to be realized at the same time and have to be disjoint (in order to avoid short circuits). However, two nets that are part of different patterns, are never realized at the same time and can share routing resources. This last property is used to minimize the routing cost of a TCON by maximizing the overlap among different patterns.

The pseudo code of the heuristic algorithm is shown in Figure 5. The outer for-loop loops over all patterns of the TCON. The inner for-loop iterates over all nets in the current pattern and routes them using a net router. The net router is a heuristic that searches a minimum cost routing tree for a given net. The same net router is used as in VPR [1] .

In order to apply the resource sharing rules, the costs of the nodes are manipulated within the TCON router. After routing a net, the cost of the resources used by that net are set to infinity. So that the next net will avoid resources that are already used by previously routed nets of the current pattern. After routing a pattern, the cost of all the resources that are already used by the current TCON are set to zero to stimulate overlap between patterns. After routing the full TCON, the resource costs are reset to the negotiated congestion cost.

2.4. Limitations of TROUTE

TROUTE has two limitations. First, TROUTE is not able to route complex patterns. 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 [8] . Thus, when patterns become too complex the TCON routing heuristic can in many cases not find a routing graph.

Second, routing a TCON scales exponentially with the complexity of that TCON. In the 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 has 8 patterns (see Figure 2) 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 re-

[1]The cost of a subgraph is equal to sum of the costs of the routing resources it contains.

Table 1: The connection representation of the 2×2 crossbar switch

Connection	Condition
(i_0, o_0)	$C_D = \neg(p_0 \vee (p_1 \wedge p_2))$
(i_0, o_1)	$C_F = p_1 \wedge (\neg p_0 \vee p_2)$
(i_1, o_0)	$C_C = p_0 \wedge (p_1 \vee p_2)$
(i_1, o_1)	$C_E = \neg p_1 \wedge (p_0 \vee \neg p_2))$

ducing 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 smaller TCONs each representing a 4:1 multiplexer for each output of the 4×4 crossbar. The patterns of these TCONs contain only one net, and thus the obstacle avoidance problem is avoided. Additionally, each TCON contains only four patterns, which reduces the routing complexity. The process of finding a 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.

3. A CONNECTION-BASED REPRESENTATION

In order to solve the problems of TROUTE, a new routing algorithm is proposed in [9], the connection router. The most significant difference with TROUTE is the internal representation of the parameterized interconnections in the tunable circuit. The Connection router uses a unique connection-based representation instead of the pattern-based representation in TROUTE . The connection-based representation represents a tunable circuit as a set of (T)LUTs and connections. Each connection is 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 2, is listed in Table 1. The connection-based representation is more scalable than the pattern representation. . 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 once using TROUTE.

3.1. The Connection Router

The pseudo-code of the connection router can be found in Figure 6 . The negotiation loop of the Connection router rips up and reroutes connections instead of TCONs. Dijkstra's algorithm is used to calculate the lowest cost path between the source and the sink of the connections.

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

Fig. 6: Pseudo code for the negotiated congestion loop of the Connection router.

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 simply counts the number of nets that were sharing the node, but now things are more complicated. The difficulty lies in the fact that under certain circumstances, connections are allowed to share resources, which is not the case for nets in PATHFINDER and TCONs in TROUTE. Connections are allowed to overlap if they carry the same signal or if they are not active at the same time. In order to find the occupation (and the congestion) of a node the Connection router has 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 [7] . This problem has to be solved in the inner loop of the routing algorithm and leads to exuberant runtimes.

3.2. Simplified Partitioning

In [9] an approximation of the congestion of a node is proposed. It only allows overlap between connections that either share the same source or the same sink;

- Connections that share the same source are allowed to overlap, because they carry the same signal.
- Connections that share the same sink are allowed to overlap, because they are never active at the same time, at least if the input to the router is assumed to be a legal tunable circuit (no shorts).

These simplified overlap rules allow a heuristic approach of the minimum clique cover problem. In this approach only two partitionings are considered: the partitioning according to the sources and the partitioning according to the sinks. The partition according to the sources divides the connections that use the node in bundles. Each bundle contains connections that have the same source. In the same way the partition according to the sink divides the connections in bundles that contain only connections with the same sink. Depending on whether the source or the sink partitioning is minimal, the congestion of a node is then approximated by the number of different sources or the number of different sinks in the set of connections that share the node, respectively. These approximations can be calculated a lot faster than the exact solution of the clique cover problem (NP complete) and thus lead to a large reduction in runtime.

The Connection router minimizes the cost of the solution, which is the sum of the costs of each of the nodes used in the solution. Therefore, it is important for the Dijkstra algorithm to accurately assess how a node used in the path for a connection contributes to the cost of the complete solution. If a node is legally shared between several connections this node should only partially contribute to the cost of the path, so that the sum of all the path costs equals the cost of the complete solution. The cost of a node is thus given by:

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

where $share(n)$ is the number of connections that legally share node n with the connection that is being routed. It is easy to see that 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.

3.3. Quality of the Connection Router Solutions

The simplified overlap rules do not cover all routing resource sharing possibilities. This affects the quality of the routing results. The solutions contain up to 29% more wires. Another downside is the execution time. The connection router needs a factor 6 or more time than TROUTE to find a solution. In Section 5 the performance of the connection router is discussed in more detail.

4. MAXIMIZING RESOURCE REUSE BETWEEN CONNECTIONS

To improve routing quality, the routing resource sharing rules are expanded. One possibility is to compare connection conditions at run-time. Connections are routed in the inner loop of the routing algorithm in Figure 6. The nodes in the RRG are explored and the node costs are calculated. To calculate the node cost, the connection conditions need to be set side by side. So the comparison is needed in the kernel of the routing algorithm. This leads to very long execution times.

A new approach is to decide which connections may share routing resources before the negotiated congestion loop begins. In this new approach, connection conditions only need to be compared once at the beginning of the routing algorithm. In our approach we use a heuristic, were we partition the connections in bundles. The connections are then partitioned in bundles. A bundle is a set of connections that may share resources. The new problem that needs to be faced is how and which connections are bundled.

```
while congestedResourcesExist () :
  for each bundle $\beta$
    $\beta$.ripUpRouting()
    for each connection $\zeta$ in $\beta$ do:
      $\zeta$.path = dijkstra ($\zeta$.source, $\zeta$.sink)
      $\zeta$.resources.updateCongestionCost()
      $\zeta$.resources. setCost (0)
    $\beta$.resources. resetCost ()
  allResources () . updateHistoryCost ()
```

Fig. 7: Pseudo code for the negotiated congestion loop of the Connection Bundle router.

4.1. Merging Bundles

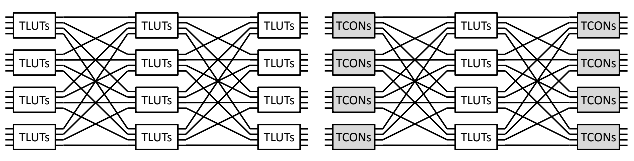
In the new router bundling starts from the simplified resource sharing rules. The simplified resource sharing rules of the Connection router in subsection 3.2 imply two different partitionings. The connections are bundled in sink bundles and in source bundles. Source bundles are sets that contain connections that have the same source and sink bundles are sets that contain connections with the same sink. This simplified partitioning is now expanded by merging bundles. Two bundles can share routing resources if all the connections of the first bundle are allowed to share resources with all the connections of the second bundle. For example, a connection of a sink bundle can share resources with a connection of a second sink bundle if they share the same source or if they are not active at the same time. To check this, connection conditions need to be compared.

4.2. Connection Bundle Router

To route bundles of connections, the routing algorithm in Figure 6 only needs to be adapted slightly. The pseudo code for the adapted routing algorithm can be found in Figure 7. The congestion loop of the Connection Bundle Router contains two nested for-loops. The outer loop loops over all bundles. The inner loop loops over all connections in the current bundle. In order to stimulate resource reuse in one bundle the cost of the resources of other connections in the bundle are set to zero in the inner loop.

5. EXPERIMENTS AND RESULTS

The reconfiguration-aware routers are compared and validated when routing Multistage Interconnect Networks that are known as Clos Networks [5] . The Clos network used for validation and comparison, uses 4×4 crossbar switches as building blocks. The 4×4 crossbar switches are used because these can be efficiently implemented with four 4-input TLUTs or four TCONs, as described in Section 2.4 .



(a) TLUT implementation (b) TCON implementation

Fig. 8: The two implementations of the 16×16 Clos network, that use reconfiguration to control the crossbar switches

A simple FPGA architecture[2] with logic blocks containing one 4-LUT and one flip-flop is used. 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 ($cols$), rows ($rows$) and the number of wires in a routing channel (W). Although we test on a simple FPGA architecture, the routing algorithms explore FPGA architectures via an RRG and that makes them inherently generic.

5.1. The Benefits of Reconfigurable Routing

In this section the results of the Clos switch experiment demonstrate the usefullness of parameterized interconnections. Three implementation types, called *Conv*, *Tlut*, *Tcon*, each for three sizes: 16×16 (3 stages), 64×64 (5 stages) and 256×256 (7 stages) are compared. *Conv* uses signals to control the crossbar switches while *Tlut*, *Tcon* use reconfiguration. *Tlut* only uses reconfiguration of LUT truth tables while *Tcon* uses both reconfiguration of LUTs and reconfiguration of the interconnect network. In *Tlut* all the crossbar switches are implemented with 4 TLUTs while in *Tcon* the crossbar switches in the even stages are implemented using TLUTs and the crossbar switches in the odd stages are implemented with reconfigurable interconnections. This last implementation results in a good balance between TLUTs and reconfigurable routing.

The routing of the *Conv* and *Tlut* implementations is done with the VPR routability-driven router. Their placement is done using the VPR routability-driven placer with default settings. The placement of the TCON implementations is done using an adapted version of the VPR routability-driven placer, called TPLACE (beyond the scope of this paper). Each TCON implementation is routed with three reconfiguration-aware routers;

- *Troute**: The pattern representation of the tunable circuit is manually optimized, as described in subsection 2.4 and routed with TROUTE

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

Table 2: Properties of six multi stage Clos network implementations for the different methods, Conv, TLUT and TCON. The TCON implementations have been routed using three different routing algorithms.

Size	Type	Area		Architecture				Time(s)
		LUTs	Wires	W_m	W	Rows	Cols	
16	Conv	202	2131	6	7	20	20	7.96
	TLUT	48	526	4	5	10	10	1.06
	TCON – *Troute**	16	456	6	7	8	8	0.45
	TCON – *Con*	16	428	5	7	8	8	1.90
	TCON – *Bundle*	16	355	5	7	8	8	0.20
64	Conv	1016	13613	6	7	47	47	294.73
	TLUT	320	3511	8	10	23	23	24.71
	TCON – *Troute**	128	4178	11	13	18	18	24.37
	TCON – *Con*	128	4372	11	13	18	18	45.32
	TCON – *Bundle*	128	3328	10	13	18	18	10.01
256	Conv	6760	97994	9	11	114	114	15415.51
	TLUT	1792	25353	13	16	53	53	1234.66
	TCON – *Troute**	768	24851	19	24	39	39	1011.63
	TCON – *Con*	768	35229	20	24	39	39	6398.97
	TCON – *Bundle*	768	21793	15	24	39	39	1233.76

- *Con*: The connection representation is routed with the Connection router with simplified overlap rules

- *Bundle*: The connection representation is routed with the Connection Bundle router

For each implementation the number of LUTs, the minimum channel width (W_m) and the number of wires of a low-stress routing is measured. Table 2 shows the results and the parameters of the FPGA architecture. A low-stress routing [1] is assured by choosing the number of wires per channel 20% larger than W_m, the minimum channel width (W), because this is typical routing problem. FPGA manufacturers normally build enough routing into their FPGAs that average ciruits have some spare routing available.

The TCON implementation saves a factor of 7.9 up to 12.6 in the number of LUTs compared to the *Conv* implementation and up to a factor of 3 compared to the *Tlut* implementation. A factor of 4.1 up to 6 can be saved in the number of wires compared to the *Conv* networks and up to a factor of 1.5 compared to *Tlut*. Because the LUTs get placed closer together W_m goes up for the 64×64 and the 256×256 TCON implementations.

The Connection Bundle router (*bundle*) gives the best quality routing solution, with 12% up to 22% less wires than the TROUTE solutions and with 17% up to 38% less wires than the *Con* solutions. And also solutions with lower track widths can be found. This improvement is more pronounced for the larger 256×256 switch network. The minimum track

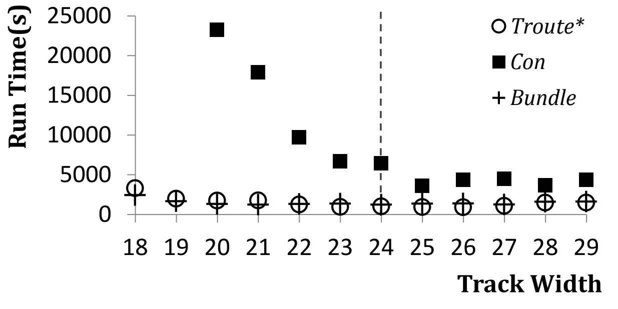


Fig. 9: The runtime to route the 256×256 Clos switch networks for a given track width for the different reconfiguration-aware routers

width found for the 256×256 network is 21% smaller than TROUTE and 25% smaller than the connection router.

5.2. Runtime vs. Track Width Trade-off

In this subsection the 256×256 clos network is used to show the runtime vs. track width trade-off. The execution time of the three reconfiguration-aware routers *Troute**, *Con* and *Bundle* are plotted in Figure 9. Note that the extra design time needed to optimize the pattern representation, in case of *Troute**, as described in subsection 2.4, is not accounted for in the chart. The timing experiments are executed with an Intel Core 2 processor running at 2.13 GHz with 2 GB of memory running the Java HotSpot^{TM} 64-Bit Server VM.

A general trend is that routers need more time to solve a routing problem as the track width of the FPGA gets smaller. The main cause is that more congestion occurs. The negotiated congestion mechanism needs more iterations to solve this congestion.

Although there is a general trend, the different reconfiguration-aware routing algorithms scale differently. If the 256×256 clos network is routed on a FPGA with track width 24, then *Bundle* finds a routing solution a factor 5.2 faster than *Con* (see Table 2). This is mainly due to extending the resource sharing possibilities. More routing resource sharing means less congestion, so the runtime drops. As the track width becomes smaller, this gap in runtime between *Con* and *Bundle* increases. The Connection Bundle router needs about the same amount of time to route circuits as *Troute**.

6. CONCLUSION

In this paper the Connection Bundle router is introduced, an automated method to route a circuit with parameterized interconnections. The Connection Bundle router uses the scalable, unique connection-based representation for the reconfigurable interconnections and automatically bundles the connections that may share resources. It produces routing solutions in less time (up to a factor 5.2) and with a better quality in terms of number of wires (up to 38%) and minimum track width (up to 25%) than its predecessors, TROUTE in [4] and the Connection router in [9] .

Acknowledgment

This work was supported by the European Commission in the context of the FP7 FASTER project (#287804).

7. REFERENCES

[1] V. Betz, J. Rose, A. Marquardt, V. Betz, J. Rose, and A. Marquardt. Routing tools and routing architecture generation. In *Architecture and CAD For Deep-Submicron FPGAS*, volume 497 of *The Kluwer International Series in Engineering and Computer Science*, pages 63–103. Springer US, 1999.

[2] K. Bruneel, W. Heirman, and D. Stroobandt. Dynamic Data Folding with Parameterizable FPGA Configurations. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(4):43:1–43:29, 2011.

[3] K. Bruneel, F. Mostafa Mohamed Ahmed Abouelella, and D. Stroobandt. Tmap: A reconfigurability-aware technology mapper. In G. Jacquemod, C. Luxey, and J.-P. Damiano, editors, *Design, Automation and Test Europe: University Booth*, Nice, 4 2009.

[4] K. Bruneel and D. Stroobandt. TROUTE: a reconfigurability-aware FPGA router. In *LECTURE NOTES IN COMPUTER SCIENCE*, volume 5992, pages 207–218, Berlin, Germany, 2010. Springer Verlag Berlin.

[5] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, XXXII:406–424, 1953.

[6] K. Heyse, K. Bruneel, and D. Stroobandt. Mapping logic to reconfigurable fpga routing. In *Field Programmable Logic and Applications (FPL), 2012 International Conference on*, aug. 2012.

[7] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[8] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, FPGA '95, pages 111–117, New York, NY, USA, 1995. ACM.

[9] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. A connection router for the dynamic reconfiguration of FPGAs. In *LECTURE NOTES IN COMPUTER SCIENCE*, volume 7199, pages 357–365, Berlin, Germany, 2012. Springer Verlag Berlin.

[10] M. J. Wirthlin and B. Hutchings. Improving Functional Density Through Run-Time Constant Propagation. *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, pages 86–92, 1997.