How to Shrink My FPGAs

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date: 19. Feb. 2022
How to shrink my FPGAs
—
Optimizing Tile Interfaces and the Configuration Logic in FABulous FPGA Fabrics

King Lok Chung, Nguyen Dao, Jing Yu and Dirk Koch
The University of Manchester, UK
{Nguyen.Dao, Jing.Yu, Dirk.Koch}@manchester.ac.uk

Abstract—Commercial FPGAs from major vendors are extensively optimized, and fabrics use many hand-crafted custom cells, including switch matrix multiplexers and configuration memory cells. The physical design optimizations commonly improve area, latency (=speed), and power consumption together. This paper is dedicated to improving the physical implementation of FPGA tiles and the configuration storage in SRAM FPGAs. This paper proposes to remap configuration bits and interface wires to implement tightly packed tiles. Using the FABulous FPGA framework, we show that our optimizations are virtually for free but can save over 20% in area and improve latency at the same time. We will evaluate our approach in different scenarios by changing the available metal layers or the requested channel capacity. Our optimizations consider all tiles and we propose a flow that resolves dependencies between CLB and other tiles. Moreover, we will show that frame-based reconfiguration is, in almost all cases, better than shift register configuration.

I. INTRODUCTION

The major FPGA vendors have large design teams with experienced engineers for designing extensively optimized FPGA fabrics which deliver high quality of results in terms of density (=area), latency, and power for FPGAs. The optimizations include a complex interplay between the architecture and the physical implementation to maximize customer utility and requirements.

To illustrate the complexity of designing FPGAs (or embedded FPGAs – eFPGAs), Figure 1 shows the FABulous ecosystem [1] that itself integrates several industry and open-source tools to generate the physical FPGA ASIC implementation as well as a flow to compile user circuits into bitstreams. Fabulous provides flows for physically implementing an FPGA fabric on a chip and for parametrizing the SymbiFlow [2] open-source tool suite that is integrating Yosys [3], nextpnr [4], ABC [5] and VPR [6] that is used to compile Verilog to bitstreams for the FPGA fabrics. Moreover, FABulous integrates BitMan [7] for the bitstream assembly and the Verilator [8] for simulation. The framework is very versatile and, for instance, for the ASIC implementation path, it can (besides some industry tools) use the OpenLane [9] tool suite, which together with the Skywater 130nm process, provides an entirely open design environment including the ASIC process design toolkit (PDK). The open Skywater 130nm process [10] is currently very popular in the open-source hardware community thanks to Google’s OpenShuttle program that resulted in about 80 MPW projects over the last six months [11].

In this paper, we are using FABulous to generate FPGA tiles and Yosys (vers. 0.9+4052) for logic synthesis. Only for the physical implementation of the FPGA tiles, we will use an industry tool (Cadence Innovus 15.2) as it still provides better, more relevant results than OpenLane in its present version.

This paper deals mostly with the physical optimization of tiles consisting of primitives and routing resources that form the basic building blocks for stitching together an FPGA fabric. This paper is built on the FABulous eFPGA framework that aims at providing high-quality FPGA fabrics as close as possible to the gold standard. To assist this, we are adding physical optimizations for improved density, latency, and possibly power. The paper contributions include:

• Configuration cell remapping. In Section IV, we will modify the relative position of configuration memory cells in both the fabric netlist and the physical implementation with the goal to minimize congestion inside...
the logic tiles of an FPGA fabric.

- **Tile interface layout optimizations.** In Section V, we will optimize the tile interface. This means that we define the exact physical connections of a logic tile with the aim to shrink its area.

- **Heterogeneous tile optimizations.** In Section VI, we will optimize DSP and register file tiles.

- **Study on frame-based versus shift-register configuration modes.** Section VIII examines the different cost factors and features for providing an optimal infrastructure to write configuration data into the fabric.

This paper is a contribution to the open-hardware movement to promote reconfigurable technology. For our work, we used an ecosystem that is mostly open, and all finding from this paper will be available under: https://github.com/--blinded--/

## II. BACKGROUND AND RELATED WORK

As identified in the seminal textbook *Architecture and CAD for Deep-Submicron FPGAs* by Betz and Rose [12], there exist three major optimization loops in an FPGA ecosystem (see also Figure 1):

1. **Physical optimization of the FPGA ASIC.** This includes optimizations like full custom cell design for configuration storage and multiplexing, buffer and wire sizing, and exploring many physical constraints. The goal here is to provide the best performance and power at the lowest cost (=area).

2. **Fabric architecture optimization.** This includes the optimization of primitives (e.g., LUTs) and the routing fabric consisting of switch multiplexers and wires. Here, the goal is to design a fabric that is suited to implement user circuits.

3. **User design optimization,** which is tweaking user circuits to make best use out of a given fabric. This is the domain of a typical FPGA application engineer.

These optimization loops do interact heavily with each other. For instance, different fabrics will impact the physical ASIC implementation and the ability of the FPGA CAD tools, and the user circuit optimization effort relates to the number of FPGA resources (and ultimately the implementation cost).

This paper is mostly focusing on physical implementation aspects of FPGA fabrics. This research has a long history. For example, custom cells were deployed in [1], [13]–[15] to improved design metrics, or the work in [16] discussed the effect of floorplanning on FPGA performance (latency). [17] discusses if pass-gates or transmission gates are better suited for building custom multiplexers for FPGAs, and customizing and optimizing the architecture of FPGAs are studied deeply in [18], [19].

In this work, we will investigate if a remapping of configuration bits can improve area, which is not covered in literature so far. Furthermore, we are tailoring the optimization of interfaces to the specific needs to build FPGA tiles.

<table>
<thead>
<tr>
<th>type</th>
<th>height</th>
<th>width</th>
<th>area</th>
<th>utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>std cell</td>
<td>CLB</td>
<td>219µm</td>
<td>219µm</td>
<td>47,961</td>
</tr>
<tr>
<td></td>
<td>REG</td>
<td>219µm</td>
<td>214µm</td>
<td>46,866</td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>443µm</td>
<td>185µm</td>
<td>81,955</td>
</tr>
<tr>
<td>custom cell</td>
<td>CLB</td>
<td>215µm</td>
<td>215µm</td>
<td>46,225</td>
</tr>
<tr>
<td></td>
<td>REG</td>
<td>215µm</td>
<td>217µm</td>
<td>46,655</td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>435µm</td>
<td>188µm</td>
<td>81,780</td>
</tr>
</tbody>
</table>

### III. CUSTOM CELLS

Most academic works (e.g., [20]–[23]) and even some industry solutions [24] do only use standard cell designs for their FPGA fabrics. Standard cell fabrics are easier to port to different process nodes, but come with poorer performance and density. However, because FPGAs require a large number of multiplexers and corresponding configuration memory cells, it is promising to optimize these two components. In the Soft++ eFPGA work [15], the authors refer to these as tactical cells, meaning that just optimizing a small number of cells can gain substantial area savings (e.g., 25% in [15]).

The OpenFPGA framework [13] provides for some processes custom multiplexers (Mux4, Mux2) and dedicated configuration D-flip-flops. FABulous provides custom multiplexers for TSMC 180nm and Skywater 130nm processes but currently has no optimized configuration cells. The reason for this is that FABulous can use frame-based reconfiguration where the configuration data is stored in latches rather than expensive D-flip-flops as needed for shift-register configuration in OpenFPGA. For instance, the custom configuration cells provided with OpenFPGA are not smaller than standard cell latches. This means that FABulous uses about the same area for configuration storage per cell than openFPGA, without the need (and risk) to design a custom configuration cell.

To examine the impact of custom cells in FABulous, we took the CLB logic, register file tiles (REG) and the DSP tiles from the FABulousSky git repository [14], and we implemented them in the smallest possible bounding boxes. For this, we used the provided netlists and we started optimizing the CLB logic tiles first by trying a number of different square bounding boxes to implement a CLB tile. The numbers reported are for the smallest successful implementation. Then we took the height of that CLB tile to implement the register file (REG) and multiplier (DSP) tiles. Note that the DSP tiles are double the height of a CLB tile plus a 5µm gap that we have between stitched tiles inside a fabric. For the REG and DSP tiles, we tried different widths to explore the smallest bounding box that will result in a successful implementation. We repeated the process, but this time used the custom multiplexers as provided by FABulous.

The benchmark CLB provides 8 Lattice-like LUT-4 with a routing fabric that was adopted from a Spartan-3 routing
The results are listed in Table I. For the used Skywater 130 process, FABulous provides only a custom Mux4 which is 6.44 $\times$ 2.72$\mu$m$^2$ versus 12.42 $\times$ 2.72$\mu$m$^2$ for the standard cell Mux4. In the standard cell design, all Mux4 contributed to the area 376 $\times$ 12.42 $\times$ 2.72$\mu$m$^2$ = 12,749$\mu$m$^2$ or 26.6% of the CLB area. This area can be shrunk by using custom multiplexers that save 48% of the Mux4 area. Ergo, the savings could be up to 6135$\mu$m$^2$. However, our experiments showed that we could only save about one-quarter of that (1736$\mu$m$^2$), and that the use of custom multiplexers is decreasing utilization instead. The trend that we found for the CLB tile also holds for the register file (REG) and DSP tile: replacing the standard cell multiplexers with custom counterparts is only marginally improving area while the core utilization of the tile is going down. Note that the tile implementation leaves a fence of 20$\mu$m at the boundary of the bounding box unused for placing the cells in order to fight congestion at the border of the tile that arises from the strict tile bounding box definition. This causes some distortion to the utilization numbers that are listed in Table I (which denote the average utilization over the full tile area).

These experiments indicate that the physical implementation is bound by wire congestion rather than the available area for mapping the cells on the chip. This particular experiment shows that providing an optimized custom multiplexer is barely showing enough improvement to justify the effort. We will therefore target optimizations that fight congestion in the following in order to take full advantage of custom multiplexers.

IV. CONFIGURATION CELL PLACEMENT

The netlist comprising the primitives and switch matrix multiplexers of a tile defines a place and route problem. In addition, a tile usually requires hundreds of configuration bits (about 500-1000 bits for a typical CLB) that form their own netlist and corresponding place and route problem. In frame-based reconfiguration mode, that configuration bit netlist is defined by select (frame-strobe) and data (frame-data) wires. In shift-register configuration mode, it is the sequential order of configuration bits and the configuration clock tree that define a netlist. The two netlists are connected by the outputs of the configuration bit cells that control multiplexers and primitives. Even these configuration bit output wires are not timing critical, they will still create a force during place and route of the tile when the primitives and routing fabric have to be implemented together with the configuration bit cells. Therefore, the mapping of configuration bits into frames and frame positions (for frame-based configuration - FBC) or the order inside a configuration shift-register (for shift-register configuration SRC) can impact area, performance, and power figures of a tile.

While it is correct to say that configuration bits can be arbitrarily remapped, support for partial reconfiguration, however, may benefit from an optimized bitstream encoding. For instance, the fewer frames a LUT function requires to be written to the FPGA, the faster that LUT could be reconfigured individually [26]. Or it is desirable to separate routing resources from primitives when using distributed memory (i.e. when LUTs can be used as small memories or shift registers). Moreover, the configuration mapping may impact power consumption during partial reconfiguration [27]. However, even the market leader Xilinx is mostly omitting these partial reconfiguration-related bitstream encoding constraints in their fabrics, and we will not further discuss the impact of the bitstream encoding on partial reconfiguration in this paper.

A. Algorithm and Implementation

To improve the congestion and the physical implementation, we take the EDA tool’s report files (which contain coordinates of all primitives) and use that information to optimize the tile netlist (at technology mapped level). The optimized netlist file is then fed back into the EDA tool to revamp a better and denser placement. We iterate through this process for a number of times to further improve the tile or give up after 100 iterations.

1) Frame-based Configuration - FBC: In an SRAM memory, we use word lines to select some data from a memory array and bitlines that carry the actual data. As shown in Figure 3a) Frame-based reconfiguration works similar to that and the bit lines (frame_data) are fed from a frame register. This register forms a wide configuration data word that may
For mapping the configuration cells to grid points, we build a matrix denoting the Manhattan distance between each of the memory cells and the grid locations while assigning each of the memory cells a coordinate. We then use the Google Optimization tool (OR tools [28]) to solve the assignment problem on that matrix. The used mixed integer programming solver minimizes the distance between each of the memory cells and the grid locations while assigning each of the memory cells a coordinate. An illustration of the algorithm’s operation is shown in Figure 5. The example shows that the algorithm can handle clusters of configuration latches by spreading out to grid points in the neighborhood, which, in turn, may require some latches to be mapped to further distend grid positions. However, this ensures that we always find a feasible solution (as long as we have at least as many grid points as we have configuration latches).

2) Shift register Configuration - SRC: As shown in Figure 3b), in SRC, the configuration cells are daisy-chained inside each tile and tile-by-tile. There are several variants to implement this approach. The long serial shift register can be split into multiple parallel and correspondingly shorter chains for improving configuration speed. Some approaches include bypass logic that allows skipping entire tiles from the shifting. This can improve configuration speed and most important it reduces power consumption as it can reduce dynamic power from potentially thousands or millions of configuration bits toggling at each configuration clock cycle. In this paper, we are focusing on single long baseline shift registers.

SRC is easier to implement as FBC as it uses less tile-to-tile routing resources and does not need a frame register or a decoder for frame strobe signals. However, the configuration D flip-flop cells (DFFs) are more expensive than latches and the configuration clock has to be routed with skew in mind. A detailed comparison FBC versus SRC will be provided in Section VIII.

As described in the introduction to this section, we can reorder the configuration DFFs inside the configuration shift register to improve the physical implementation of the tiles (see Figure 2). Computing this order is computing a (minimum) Hamiltonian path, which is a well-examined problem. We ran experiments using the OR tools [28] library which gave us some improvements but not as good as hoped for. Instead we found that we can adapt the grid-point mapping from FBC by daisy-chaining the configuration memory cells along the frame strobe signals to form a meander-shape that is traversing down and up the original frame strobe signals, as shown in Figure 4c).
improve further. The improvement is shown in Figure 8. As can be seen, most of the optimization potential was reached after 50 optimization runs (which corresponds to about 6 hours execution time).

2) The Impact of Metalization Layers: For our experiments, we used the 130nm Skywater process, which provides 5 metal layers. However, so far, we used only 4 metal layers for the implementation of our tiles. It is common practice for embedded FPGAs (eFPGAs) to reserve one metal layer for the top level integration. This is needed for connecting clock trees, power lines, and in many cases, the system that is surrounding the eFPGA will have some signals that have to route across the eFPGA. Leaving one metal layer reserved is the default mode in FABulous to provide users with a robust design experience.

Nevertheless, if we would use the FPGA fabric as a standalone version instead of an eFPGA, we can use the top metal layer to support the physical implementation of the fabric. To measure the impact of this on density, we repeated the experiment for the FPGA tile with all metal layers enabled while running our optimization loop. With one extra metal layer, the size of the tile can be further reduced to 190 × 190μm², which is another 3.2% improvement.

As a reference, we repeated the experiment by using all metal layers but without our optimization. In this case, the FPGA tile size reduced to 202 × 202μm² = 40,804μm² versus 37,249μm² when sparing one metal layer but running our optimizations. This shows that bitstream remapping helps substantially more for improving density than the extra metal layer. The cost of sparing one metal layer is 3.2% in density and this is the cost paid to greatly simplify the integration of an FPGA fabric into an SoC. Reserving one metal layer will in most cases allow stitching FPGA fabrics together from fully implemented hard macros (GDS files). This also means that the optimization effort that we spend on optimizing a tile can benefit multiple chips. We can therefore conclude that this optimization comes virtually for free.

3) Impact of Routing Channel Capacity: In the previous paragraph, we examined how the available metal layers impact the tile implementation. In this paragraph, we will examine how an increasing demand of wires impacts the physical implementation of a CLB tile.
The basic CLB used in the experiments had been designed for small FPGA fabrics. This is reflected in the fabric architecture graph by providing fewer long distance wires, which are normally needed for linking different modules together on a larger FPGA. Subsetting an FPGA architecture graph was, for example, used by Xilinx in their Spartan-3 family where the smallest devices do not provide hex wires [29]. The congestion in an FPGA fabric is related to the number of wires and their length. For instance, if, at a tile, we have 3 wires of length 4 starting in one direction, there will be $3 \times 4 = 12$ wires forming the interface between two adjacent tiles, as shown in Figure 9. This is quantified by the channel capacity that is described in more detail in Section V.

To examine the effect of adding long distance wires, we added 4 wires in each north, east, south, west direction and set the length to a distance of 4, 8, and 12 tiles. In all three experiments, we used the same adjacency and we only changed the wire span. The baseline CLB tile with FBC configuration has 507 wire connections at its border. The three experiments will add 64, 128, and 192 wires respectively to the tile.

The results are listed in Table III. We can see that the tiles became substantially larger despite that the underlying netlist is almost identical in terms of resources. This reflects the increased stress on the metal stack that is rising with the span. This means that in these experiments, the tile area is bound by the metalization resources rather than the silicon capacity to accommodate gates.

The results emphasize the importance to design a balanced architecture graph that optimizes both, the silicon density and the use of the metal stack [12]. This is in particular a concern for embedded FPGAs that are commonly more constrained with the number of usable metal layers. For instance, if the physical implementation is heavily congested, then extra multiplexing for tapping wires or using wires bidirectional may become attractive optimization options [30]. While this topic is beyond the scope of this paper, the here proposed optimizations will work orthogonal to such architectural improvements and will provide some extra gain in density virtually for free.

4) Improvement of SRC CLB Tiles: We repeated the experiments using the SRC configuration mode in the otherwise identical CLB tile. Here we run the experiments two times. In the first run, we generated the CLB in FABulous with a configuration shift register using Skywater standard cell D flip-flops (SRCFF). Because a flip-flop is about double the size of a latch, OpenFPGA suggests using a custom configuration cell instead of a DFF to offset the area overhead for their shift register configuration. However, OpenFPGA is currently not providing custom configuration cells for Skywater 130. The closest available custom configuration flip-flop is reported for a 45nm process with about 60% the size of a DFF [31]. In this paper, we mostly investigate physical implementation aspects. In order to investigate the impact of a custom configuration cell, we replaced the DFFs from the first run with latches for a second run (SRCFF). This netlist is not functional, but reassembles well the physical optimization problem when using custom flip-flops (and assuming some very good optimization levels). In both runs, the configuration clock was implemented as a separate clock tree.

The results are recorded in Table II for comparison against FBC. The standard cell version of the shift register approach (SRCFF) is 17,784 $\mu m^2$ (38.5%) larger than the corresponding tile using frame-based configuration. This is much more than what we would expect for fundamentally replacing 586 latches with D flip-flops and rewiring these cells. This is a strong indicator that the scan chain is hugely irregular and causing congestion. This observation holds correspondingly for the custom cell shift register version (SRCFF) that due substituting the D flip-flops with latches uses exactly the same core area for all the physical cells, but is still 13.4% larger than
V. TILE INTERFACE LAYOUT OPTIMIZATION

The channel capacity denotes the number of wires that cross a cut between any two tiles (see Figure 9 for an explanation). The channel capacity depends on the number of wires and their span (i.e., the routing distance in tiles). A typical CLB has a channel capacity of about 200–400 wires in both horizontal and vertical directions. In a tile-based FPGA layout, all these channel wires have to be connected to wire pins at the border of each tile. In addition, an FPGA fabric needs some global signals, like clocks and wires for configuring the fabric, which can have a significant impact on the physical implementation.

For instance, the FBC tile from the previous section uses 32 wires in the horizontal direction for distributing configuration data and 20 wires in the vertical direction for the frame strobe signals. This contributes to 10% of the 507 IO wires of a CLB tile. The tile interfaces must be designed such that stitching of different tile types is possible. This means that, for example, an output pin at the left tile border has to align with a correspondingly aligned input pin at the right tile border of each tile that will ever use stitching.

As shown in Section III our FPGA tiles suffer from congestion, and in this section, we will optimize the tile interface to generate a denser packed (better utilized) CLB tile. This problem is not entirely new, and in [32] simulated annealing was used to optimize the physical wire layout of an on-FPGA interface between a static FPGA system and partially reconfigurable modules. While the wire allocation problem is solved for a user circuit running on an FPGA rather than for the FPGA fabric itself, the problem is closely related to our tile interface optimization.

A. Algorithm and Implementation

As a starting point, we analyzed the default FABulous tile layout, which is based on sorting wires by direction and name. To some extent, this is similar to what some scan-chain insertion tools can do (like the DFT in the Synopsys Design Compiler). For a scan chain, this strategy of arranging scan flops by sorting them by name can produce good results as this commonly reflects hierarchies and the vectorization of signals. However, for optimizing a fine-grained and hugely irregular tile netlist, this approach will likely not improve congestion, why we developed a dedicated optimization method for the tile interface optimization.

As with the configuration remapping in Section IV, the interface optimization extracts data generated by the EDA tool. This includes the floorplan (with detailed placement information of each standard cell) and the tile netlist (which is mapped to standard cells). The optimization problem has two dimensions: 1) the metal layer (we use M1 and M3 for horizontal routing and M2 and M4 for vertical routing) and 2) the position of an interface wire pin along the tile border. In each optimization round, we are computing a new optimized pin placement that is implemented by modifying the physical constraint files of the tile. The optimized interface design is then fed back into the EDA tool with the aim to reduce congestion for allowing a smaller tile area. The optimization follows the flow shown in Figure 7 and instead of optimizing the bitstream mapping, we are now optimizing the tile interface.

The operation of the algorithm is shown in Figure 10. The pin allocation algorithm will consider corresponding pins on opposite sides of the tile, hence forming a pin pair. This is because an input pin at, for instance, the north border will be an output at the south border. First, we compute the shortest lengths between a pin pair and any connected primitive for the input and output side separately. Our algorithm will consider only the shortest paths. The strategy here is that in a spanning tree, we want to reach the first node and branch eventually
from there to other nodes. Then, we compute the sum of the lengths for each pin pair and sort them by their combined lengths. The longer combined paths will mapped to the higher metal layers and the shorter paths to the lower metal layers.

Next we compute the exact pin position in each metal layer. For north-south borders, we will compute the combined horizontal component and feed this into a table that is used by Google Optimization tool (OR tools [28]) to compute the pin positions. This is related to the configuration bit grid point computation but uses only one dimension here. The algorithm will automatically resolve conflicts if multiple paths want to use the same pin pair.

### B. Evaluation

We ran the experiment in the same way that we evaluated the configuration bit remapping, and we reduced the tile borders by 1µm after each successful implementation until we eventually fail to optimize further. For 120 iterations in total, the optimization took about 10 hours, and it reduced the tile size from 215 × 215µm² to 199 × 199µm², a 14.3% reduction in area. Figure 11 shows the progression of the optimization loop over the iterations. Results for applying tile IO pin optimizations for other tiles will be presented in Section VI-A.

### VI. OPTIMIZING HETEROGENEOUS FPGA FABRICS

Modern FPGAs usually provide different types of resources to support the implementation of user circuits that require memory or multiplication. Those functions are expensive to implement just from LUTs [33]. The physical implementation of a tiles FPGA fabric introduces some dependencies between other tile types, and this section is devoted to optimizing such other tiles. The dependencies include the height of the base tile and the horizontal interface as defined by the CLBs. In Section III, we reported the improvements gained by replacing standard cell multiplexers with custom multiplexers. For that, we implemented the CLBs first and propagated the achieved height to the DSP and REG (memory) tiles.

Consequently, when applying our optimizations, we have to implement the CLB firstly and propagate the height but also the (horizontal) interface constraints to the other tiles (see also Section IV-B4). In the case of the register file tile (REG), the CLB height and the left and right tile border interfaces are propagated from the CLB. In the case of a tile spanning over n multiple vertical adjacent tiles (like the DSP), n × height(CLB) + (n − 1) × tile_gap is propagated as the height, and the CLB interface is propagated to each elementary cell (e.g., two times in our DSP example where a DSP uses the height of two CLBs and provides two switch matrices to the routing fabric).

The results in Table IV are for tiles using FBC configuration mode. The numbers confirm the low improvement of our optimizations for the standard cell designs because these problems are not very congested (see also Section III. When using custom cells, congestion is rising and allows our optimizations to kick in. However, the improvements (8.2% and 10.0%) are below the numbers that we reported for the CLBs in Section IV and Section V. We believe the reason for this is that these specialized tiles provide overall more regularity inside the netlist, which helps Innovus with the physical implementation such that, for example, the remapping of configuration bits is working somewhat less effective. Nevertheless, the improvements are still significant and can be gained fully automated.

### A. Optimizing Interfaces of Heterogeneous Tiles

We applied IO interface optimization where we remap tile pins to optimize the DSP and register file tiles. This optimized the area of the register file tile (REG) from 245 × 193µm² to 225 × 193µm² and for the DSP tile from 190 × 391µm² to 173 × 391µm². Interestingly, these are almost identical.
A. Timing Improvement

Even our main objective in this paper was to improve density, a denser (=smaller) design will likely also improve latency as wire lengths get shorter. As a latency metric, we measured the latency of all tile input pins to all tile output pins. We recorded the worst-case latency as this is most relevant for improving critical path delay in a user circuit. As can be seen in Figure 13, the latency improvements are minor than the area improvements. This is expected as the wire length is only one contributor to the overall wire propagation delay. From the previous paragraph, we saw that our optimizations can improve area similar good as adding an extra metal layer. However, for latency, the additional metal layer can cut latency over the baseline by 12.4% versus 6.1% that we achieve just from optimizations.

B. FABulous versus OpenFPGA on Skywater 130

The latest Google Shuttle run [34] on a Skywater 130nm process includes both a FABulous [14] and an OpenFPGA FPGA design [25]. Both designs implement a CLB with 8 LUT-4. The OpenFPGA CLB allows fracturing the LUTs into two LUT-3, which is essentially tapping into the LUT multiplexer tree. The CLB resources compare as follows:

<table>
<thead>
<tr>
<th>Resources</th>
<th>OpenFPGA</th>
<th>FABulous</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC_DFF_opt</td>
<td>62,001</td>
<td>36,100</td>
</tr>
<tr>
<td>SRC_CFF_opt</td>
<td>52,441</td>
<td></td>
</tr>
<tr>
<td>pin swap</td>
<td></td>
<td>37,249</td>
</tr>
<tr>
<td>comb. opt.</td>
<td>26,481</td>
<td>38,416</td>
</tr>
<tr>
<td>FBC Full metal opt</td>
<td>33,853</td>
<td>-12.4%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>+3.8%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>-11.7%</td>
<td>-19.4%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>-21.9%</td>
<td>-21.7%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>+38.5%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>+34.1%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>+13.4%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>-2.7%</td>
<td>-4.8%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>-16.9%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>-2.7%</td>
<td>-4.8%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>-16.9%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>+13.4%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>-2.7%</td>
<td>-4.8%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>-16.9%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>+13.4%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>-2.7%</td>
<td>-4.8%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>-16.9%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>+13.4%</td>
<td></td>
</tr>
<tr>
<td>frame-based config</td>
<td>-2.7%</td>
<td>-4.8%</td>
</tr>
<tr>
<td>frame-based config</td>
<td>-16.9%</td>
<td></td>
</tr>
</tbody>
</table>

VII. SIDE-BY-SIDE COMPARISON

We collected the area results of our different optimizations in Figure 12. We took the default FABulous CLB tile with frame-based configuration as the baseline. We found that our present configuration bit remapping and tile pin remapping had been able to gain the same improvement in area (−19.4%). We believe that both optimizations are hitting a utilization level (about 85% core utilization) from which Innovus stops improving any further, which we are currently investigating.

We also applied both optimizations together, which improved the tile to -21.7%, the best improvement we achieved by our optimization on the baseline tile.

The figure also contrasts frame-based versus shift register modes for building an FPGA CLB tile. As can be seen, shift register configuration results in much bigger tiles, which can only be partially offset by custom configuration cells if the design is congested. Similar to the custom multiplexer optimization (see Section III), using custom configuration cells is mostly resulting in an underutilized tile. However, when rearranging the configuration bits, we can reduce the tile size close to what we achieved for FBC mode.

A. Timing Improvement

Even our main objective in this paper was to improve density, a denser (=smaller) design will likely also improve improvements that we achieved by the configuration cell remapping.

VIII. FRAME-BASED VERSUS SHIFT-REGISTER CONFIGURATION MODES

Even though shift-register configurations are easier to implement, there are multiple advantages of frame-based configuration modes over shift-register configuration, the most important include:

- **Latches for configuration storage**, which saves area.
- **Partial reconfiguration**. While SRC could be implemented using tile bypass multiplexers, this is not suited for PR because it is commonly required to configure routing resources individually (e.g., to route a static signal through a partial region). SRC would not allow this as all configuration bits are shifted together, while FBC can update each individual configuration bit separately.
- **Lower power consumption during configuration**. A CLB tile uses about 500 to 1000 configuration bits (about two orders of magnitude more configuration memory bits than the number of user flip-flops in a CLB). Therefore, shifting in a configuration may potentially toggle large numbers of bits simultaneously with a corresponding dynamic power footprint. Moreover, in shift register configuration, a configuration is only valid if it reaches its final position. However, during the configuration process, a configuration may cause many unwanted intermediate configurations like ring oscillators or even short circuits.
• Fast configuration blanking and configuration wild-carding SRC requires shifting in a blanking stream, which takes as long as the configuration. Frame-based reconfiguration normally allows writing the configuration from the frame register to multiple frames, which can implement a fast configuration multicasting (e.g., the multiple framewrite feature (MFWR) in Xilinx FPGAs [35]. This mode is available in Xilinx FPGAs and FABulous FPGAs and can be used for fast configuration blanking or some configuration compression schemes.

Table II showed that a FBC CLB tile is 3.1% smaller than a SRC CLB. A frame register costs about 30% of the width of a CLB tile, and the frame select logic 20% of the height of a CLB. This means that after 10 columns and 7 rows of CLBs, the frame-based configuration would be cheaper to implement, even considering that custom configuration cells could save half the configuration cell area (which is an optimistic estimate for the achievable optimization).

IX. CONCLUSION

This paper proposed configuration cell remapping and tile interface pin remapping as optimization techniques for shrinking FPGA tiles. Combining both techniques gained an area improvement of 21.7%. We evaluated our approach over various different scenarios, including changing the number of metal layers and examining routing fabrics that use different channel capacities. In this paper, we provide a complete optimization approach that includes all tiles, including logic, memory arithmetic, or any other specialized tile. The numbers collected in this paper represent a full week (24/7) of execution time on a large server, including failed runs, probably close to a month.

As a further contribution, we contrasted frame-based configuration mode versus shift-register configuration mode. Our results show that frame-based configuration is winning in all aspects for any realistic fabric size.

While we used the FABulous FPGA framework to run our experiments, there is no fundamental obstacle for users to integrate our optimizations into other frameworks (e.g., OpenFPGA). For OpenFPGA, this holds in particular as both frameworks use the same ASIC backends (Innovous). To support this, we released our optimization tools under: blinded

With more than 20% improvement in area, our results are better than what we had expected when starting this project. This improvement was made possible by reversing the physical optimization loop as shown in Figure 1. Normally, a model is refined over multiple abstraction levels to produce the physical ASIC implementation in a strict top-down manner. In our flow, we modify the physically implemented netlist at the bottom and propagate the outcome back to the model (e.g., in the form of a bitstream remapping table). While this approach may not work well for general ASIC design, it is an interesting avenue for future research for implementing FPGA fabrics.

With this work, we contribute to the growing open-hardware community and empower users to integrate high-quality FPGAs (and reconfigurable fabrics in general) into their designs. For instance, the RISC V movement has created much interest in (open) hardware design, and eFPGAs can add exciting features to corresponding systems (e.g. [36]). This integrates into a larger ecosystem of open-source projects, such as SymbiFlow [2], Yosys [3], nextpnr [4], VPR [6], OpenRAM [37], OpenLane [9] and the Verilator [8], but also the open PDK from Skywater [10]. Our next step will include replacing Innovus with OpenLane to provide an open-everything FPGA experience with the aim to provide a quality of results close to or, perhaps, even better than what industry tools achieve due to customization and optimizations.

REFERENCES
