Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-022-01786-z

V-FPGAs: Increasing Performance with Manual Placement, Timing Extraction and Extended Timing Modeling

Johannes Pfau[1] · **Peter Wagih Zaki**[2] · **Jürgen Becker**[1]

Received: 1 October 2021 / Revised: 26 February 2022 / Accepted: 20 June 2022

Abstract
Virtual FPGAs (V-FPGAs) are used as vendor-independent virtualization layers, to retrofit features which are not available on the host FPGA and to prototype novel FPGA architectures. In these usecases, the achievable clock frequencies of V-FPGA user applications are a major concern. The abstraction layer inherently induces overhead, but this aspect is reinforced by nonuniformity effects: When V-FPGA cells perform worse locally, basic architecture modeling generalizes these worst-case path delays to the whole device, limiting applications to a lower frequency than theoretically achievable. We propose three approaches to attenuate these effects: First we introduce uniformity metrics and manual V-FPGA placement strategies for more uniform placement, improving achievable frequency by 16 %. Second, we propose a framework for automated timing extraction, enabling individual characterization of each V-FPGA design. Third, after evaluating Vivado synthesis strategies, we extend the timing model for non-uniform timings, achieving improvements of up to 28 %.

Keywords FPGA · EDA · Placement · Virtual FPGA

1 Introduction

In recent years, virtual Field Programmable Gate Array (FPGA) architectures (V-FPGAs) have been been introduced in academia [1]. Unlike common commercial and academic FPGAs, the V-FPGA is an FPGA architecture layered onto a base FPGA architecture: A commercial host FPGA architecture is synthesized for a silicon chip target, and the V-FPGA layer is synthesized for that host FPGA architecture. The virtual layer is implemented as a bitstream to be programmed onto the host FPGA. User applications are synthesized using a custom toolchain for the V-FPGA layer and the resulting application bitstream is programmed onto it. V-FPGA architectures have been applied for three main use cases: First, as an abstraction layer, providing a common bitstream format independent of commercial FPGA architectures. This allows using features such as partial dynamic reconfiguration on FPGAs which do not natively support this [2]. Second, V-FPGAs can be used for FPGA architecture research: Novel ideas can be integrated in the architecture and tested on a hardware implementation, which can provide additional insight compared to simulation. This becomes especially useful when investigating heterogeneous System-on-Chip (SoC) solutions, which may combine processing systems and reconfigurable logic [1, 3]. Third, using the V-FPGA as a basic FPGA architecture: Here it is realized using standard synthesis approaches for silicon targets and can be used to evaluate the usage of different logic cell technologies in FPGAs [4, 5].

V-FPGA architectures have to address various issues and limitations, some inherent in the very idea of a virtualization layer, such as e.g. overhead of various kinds caused by the virtualization layer. One example is area overhead of placed user applications, comparing total area required on the host FPGA including the virtualization layer to direct placement on the host FPGA. Some of the difference is caused by different synthesis tools for the host FPGA and the V-FPGA: Whereas commercial vendor tools are used for host FPGAs,

✉ Johannes Pfau
pfau@kit.edu

Peter Wagih Zaki
peterwzaki@gmail.com

Jürgen Becker
becker@kit.edu

[1] Institute for Information Processing Technologies, Karlsruhe Institute for Technology, Engesserstraße 5, Karlsruhe 76131, Baden-Württemberg, Germany

[2] German University in Cairo, El Tagamoa El Khames, New Cairo City 11835, Cairo, Egypt

Published online: 05 July 2022

V-FPGA architectures commonly use the open source tool Versatile Place and Route (VPR). Area overhead is further caused by the structures of the V-FPGA itself, especially the configuration logic. This logic, which is used to configure the application bitstream onto the V-FPGA and to store the configuration for the V-FPGA Lookup Tables (LUTs), is usually implemented in host FPGA user logic. It therefore doesn't make use of the explicit configuration logic of the host FPGA, increasing the amount of normal LUT resources used. Although making use of host FPGA configuration resources is possible, it will cause the V-FPGA architecture to resemble the host FPGA very closely, therefore impeding many of the common V-FPGA use cases. Because of this, there is a trade-off between the achievable grade of abstraction from the host FPGA and the overhead caused by the virtualization, which has to be considered carefully.

Similar considerations apply for the maximum achievable frequency of a user application targeting the V-FPGA. Whereas an application directly placed onto the host FPGA can make use of commercial vendor tools, for V-FPGAs the applications will have to be synthesized with custom tools. VPR requires an architecture description to provide information about the V-FPGA, including the logical structure of the V-FPGA, aspects such as LUT size, Configurable Logic Block (CLB) structure, how individual blocks are connected in the interconnect and more. Whereas this information is easy to obtain or already given by design decisions made during development of the V-FPGA architecture, the second class of information, the timing information, is more difficult to acquire. To properly determine the frequency a user design can be clocked at, the placement tool VPR needs to know about the delays of all logical elements of the V-FPGA architecture. These logical elements often map to multiple elements on the host FPGA. In addition, the delay encountered for one element at one location of the V-FPGA is usually different from the delay of a logically equivalent element at a different location. As an example, the delay of a LUT at location *(1, 1)* may be different from the one at location *(4, 4)* due to irregularities in the placement and routing of the V-FPGA onto the host FPGA.

To address these issues, two approaches will be considered in this publication. The first approach is making the V-FPGA placement onto the host FPGA more uniform. As the timing information in a common VPR architecture considers all elements of one type to have identical delays, the V-FPGA timing characterization has to use the worst case, e.g. the maximum delay of all elements of one type in the V-FPGA. In the example case of LUTs, the maximum delay of all V-FPGA LUTs would be used to characterize the LUT in the architecture. With this approach, the maximum observed delay is most important and outliers anywhere in the V-FPGA affect all of the V-FPGA elements. Uniformity of the design is therefore beneficial, as long as it also reduces the maximum delay. For special use cases, uniformity is furthermore not just a secondary metric, but inherently important: Use cases involving relocation of placed logic require similar delays everywhere on the V-FPGA. Furthermore, achieving quick tool runtime in VPR requires assumption of a uniform FPGA architecture, although other solutions are possible and will be considered in the following. When modeling a uniform architecture, a worst case value may be used, but it will severely limit the maximum frequency of user applications. The first part of this work therefore considers approaches to reduce the number of maximum delay outliers and increase uniformity of observed delays. This part is an extended version of our previous publication [6], which provided the initial idea of manual placement. Manual placement tries to customize the placement of V-FPGA resources onto the host FPGA, trying to achieve more uniform delays than default placement does. In addition, we follow up on this work to describe how timing information is extracted from the design and used to model the architecture in VPR. The original results have been re-evaluated to be based on a fixed track width for all V-FPGA sizes compared, avoiding confusion due to multiple parameters changing between compared designs. We also provide a new comparison of Vivado synthesis strategies and the influence those have on the uniformity of the V-FPGA.

We further introduce a new, second approach to address the described V-FPGA issues: Accepting the non-uniformity of the V-FPGA, in which we present adjustments in VPR synthesis flow to handle the non-uniformity: First we obtain the timing information for each single logic element of the placed V-FPGA. Then, instead of using the maximum value as in the basic case, we model each CLB in the VPR architecture with individual timing information. Whereas this leads to improved user application frequency, it has the drawback of being more complex to model, increasing the VPR runtime and therefore having limited scalability towards larger V-FPGA designs. For large V-FPGA designs, increasing the uniformity as shown in our first approach may prove to be more viable.

2 Related Work

Regarding previous research in placement algorithms for FPGA, we differentiate two research areas: Placement algorithms for generic FPGA applications and manually guided placement for specific applications. Research on generic placement algorithms has largely been conducted using academic open source tools. The most-widely used of these tools is Verilog to Routing (VTR) [7], which consists of three main tools: Odin II for synthesizing circuits designed in Verilog into generic LUT resources, ABC for technology-mapping those resources into architecture specific LUTs,

and VPR for packing, placement, and routing. Research led to the introduction of new techniques and algorithms for placement and packing, focusing on different cost targets such as timing driven, routing-driven or runtime-driven: [8] introduces an algorithm named adaptive range-based Simulated Annealing (ARBSA). It provides an adaptive approach to choose the neighborhood for each block according to the nets it belongs to. The result shows a 1.78X runtime speed up, 10 % reduction on wire length and 2 % reduction on the critical path with timing-driven optimization. Research in this area proposes general algorithms, trying to find good solutions regardless of the target application. Results are usually compared to the VTR algorithms, showing improvements over VPR which do not directly translate to comparisons with commercial tools [9].

The second group of research uses the concept of manual placement, ranging from guidance to completely manually placed designs. For example, [10] introduced a sea-of-gates architecture called Triptych. It aims to reduce the significant cost paid for routing in standard FPGAs, replacing the logic blocks with Routing and Logic Blocks (RLBs). RLBs perform both logic and routing tasks, allowing a tradeoff between logic and routing resources on a per mapping basis. Using manual placement made this architecture yield a logic density improvement of up to a factor of 3.5 over commercial FPGA's automatic placement. In another example, Shi et al. analyzed manual placement for their specific FPGA application [11]. It shows that using manual placement leads to a compact and optimized design with shorter nets, reducing propagation delay up to 25 %.

Whereas generic algorithms can be used to implement the V-FPGA, they have to be reimplemented in the Vivado implementation flow, as they originally target VPR. In addition, none of these algorithms takes the regular structure of the V-FPGA into account. The presented application specific manual placement methods on the other hand, do not directly translate to the V-FPGA structure. They need to be heavily modified to be usable for this application area. In order to improve net delay in the V-FPGA, we therefore investigate specific custom placement methods based on similar ideas, but explicitly considering the V-FPGA architecture and regularity.

3 Background

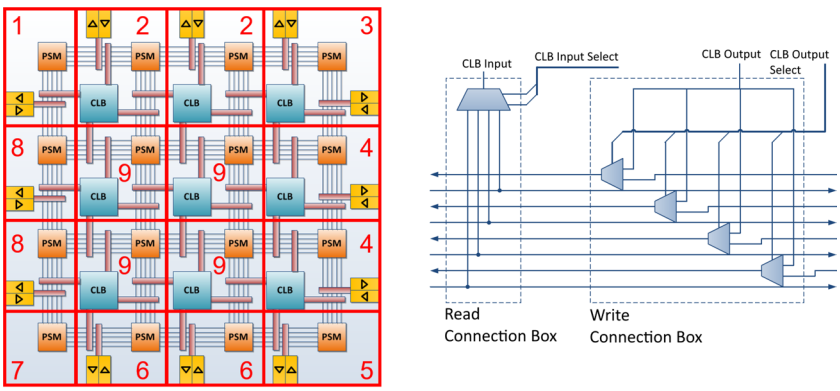
The placement strategies to be introduced make use of the regular structure of the V-FPGA. As such, they are dependent on both the V-FPGA architecture, as well as on the host FPGA architecture. The strategies have been designed to work with a certain parameter variability in these architectures, but certain assumptions have been made.

3.1 V-FPGA Architecture

Figure 1a shows the V-FPGA architecture and the arrangement into the common types (1-9) of tiles. In its simplest configuration, the V-FPGA consists of 9 different tile types, which are distinguished by orientation and contained elements. A single tile can contain all elements (like type 2 and 4), all but the I/O Block (IOB) (type 9, i.e. the central tiles), Programmable Switch Matrix (PSM) and IOB (type 1, 5, 6, 8), CLB, PSM and two IOBs (type 3), or only the PSM (type 1). Even for tiles which have the same types of elements, their differences in orientation — and therefore layout — will require them to be placed differently.

The Configuration Units (CUs) are not shown in Fig. 1a, as it is an implementation detail of the V-FPGA. They store and provide the configuration for the CLB, PSM and IOB in their respective tile and enable dynamic reconfiguration of the V-FPGA. In our V-FPGA implementation, the CU is implemented essentially as a shift register with parallel output. It will have to be mapped to the host FPGA in addition to the other components.

Figure 1a also shows the wires corresponding to relevant delays for the final VPR architecture model: Apart from



(a) 3x3 CLB V-FPGA. **(b)** CBR and CBW Connection.

Figure 1 V-FPGA architecture details: **a**: Tile Distribution and top-level architecture. **b**: CBR and CBW implementation and connection to routing channels.

intra-block delays such as delays within the CLB, these consist of nets in the global routing channels. Figure 1b shows how the CLB in a tile connects to the global routing channels. Connection points are realized using Read Connection Boxs (CBRs) and Write Connection Boxs (CBWs), which consist of multiplexers either connecting multiple wires to one CLB input or connecting the CLB output to one of the channel wires. To realize this connection, the CBW as an example consists of one multiplexer for each wire in the channel. The multiplexer selects between either forwarding the signal of the channel wire or writing the CLB output to this wire. Structurally, these multiplexers will be mapped to host FPGA LUTs. Therefore on the host FPGA, a V-FPGA wire from PSM to PSM will actually consist of at least two host wire segments and the LUT. Similar effects are also caused by IOB connections.

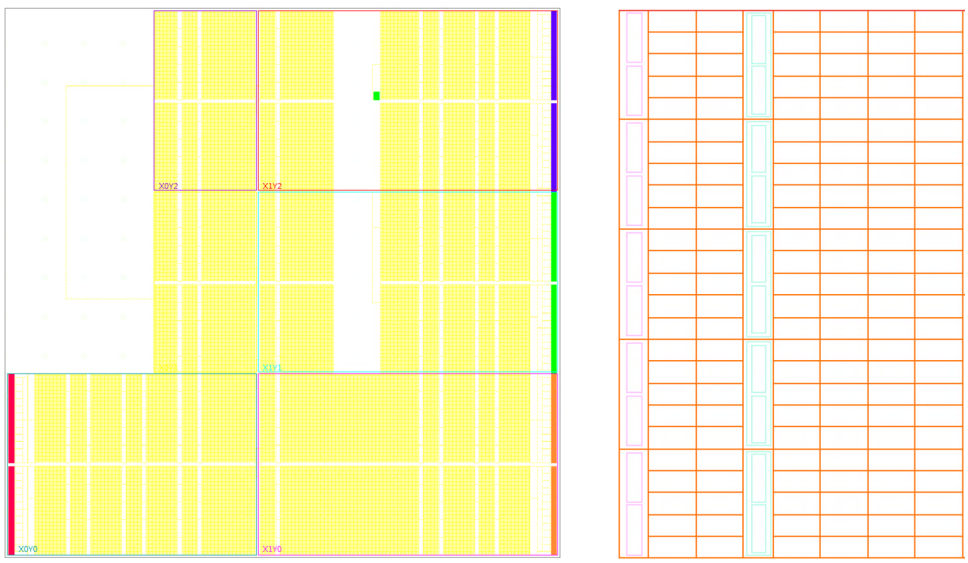
Another peculiarity of our V-FPGA implementation concerns the implementation of its bidirectional wiring: Logically, the V-FPGA architecture uses bidirectional wiring, but the implementation on the host FPGA can only make use of unidirectional wiring. Figure 1b shows how the CBR and CBW connect to different host wires, leading to different directions. In order to drive a logical V-FPGA wire in both directions, the PSMs will loop back the right-to-left signal in left-to-right direction.

All these effects cause two implications for this work: First, when constraining the design, the V-FPGA wire can not be constrained as one unit. Instead, all wire segments on the host FPGA have to be constrained individually. Furthermore, when modelling the V-FPGA architecture in VPR, the delay for the complete net is needed. For this reason the data extraction script extracts the individual segments, but for architecture modeling and for assessment of placement results in this publication, these have to be added up. In the rest of this article, we will always consider and present complete PSM-to-PSM and CLB-to-PSM paths.

3.2 Host-FPGA Architecture

Although the strategies described in this work are designed to be generic and support different host-FPGA architectures, the specific parameters used will depend on the concrete host-FPGA architecture used. For the evaluation and results shown, we focus on Xilinx 7 Series architecture and especially the XC7Z020-CLG484 FPGA. To describe how host-FPGAs architectures provide limits for achievable uniformity, we will describe the used host-FPGA architecture in detail: For manual placement, the most important aspect of the host-FPGA architecture affecting V-FPGA uniformity and delay is the host-FPGA uniformity. If the host FPGA was completely uniform, placing the V-FPGA in a uniform way would be a simple task of repeating a template structure. As shown in Fig. 2, this is not the case for recent commercial FPGAs:

Figure 2a shows the overall device layout, where CLBs are shown as small boxes. Regions of such CLBs are divided by wide gaps. These are areas where larger hard-IP blocks, such as the ARM processor cores or DDR memory controller cores, are located. Placing the V-FPGA across the whole area of this device will cause larger delays for wires connecting two V-FPGA tiles separated by hard IP, whereas adjacent V-FPGA will show reduced delay. This issue will need to be considered when finding the target area to place the V-FPGA, as well as when placing the individual tiles. Most importantly, a single tile should never be disconnected by an interspersed large hard-IP block. Furthermore, depending



(a) FPGA Layout Overview. **(b)** CLB Columns Detail.

Figure 2 XC7Z020-CLG484 Xilinx 7-series FPGA layout: **a**: Chip overview showing clock regions, peripheral blocks and large gaps caused by hard-IP such as the ARM processor cores **b**: Close-up view of CLB columns showing variation in number of adjacent CLB columns caused by interspersed BRAM and DSP hard-IP.

on the host FPGA, it may not be reasonable to use the whole area if high performance and low delays are required.

Figure 2b shows a similar, but less grave effect, in providing a close-up view of the CLB columns. Columns are also divided by interspersed small hard IP blocks, mainly Block RAM (BRAM) and Digital Signal Processor (DSP) resources. These cause the number of adjacent CLB columns to vary, an effect which needs to be considered when placing the V-FPGA tile contents and when determining the tile sizes. In addition, this limitation will cause certain variation in V-FPGA uniformity which cannot be avoided without significantly affecting other performance characteristics (e.g. using only some of the available CLBs, which would increase required area).

3.3 Metrics

To evaluate and assess the results, we first introduce several comparison metrics. The commonly used metrics — delays, uniformity and area — have a specific meaning for V-FPGA targets and are crucial for V-FPGA applications.

3.3.1 Uniformity

Uniformity is a measurement of local delay variation across the V-FPGA structure. Placing every tile in the same way, the V-FPGA is a uniform structure, which in theory could be placed uniformly on the host FPGA. As explained previously, not all tiles have exactly the same internal structure and non-uniformity of the host FPGA architecture will further degrade the uniformity. To address this, our definition of uniformity divides the V-FPGA into N_C sets, where each set represents one column C. We also group nets into classes, so that similar nets in different tiles are within a single class. We differentiate between these classes:

1. **PSM Left:** Horizonal nets, starting at PSM left output multiplexers and ending at PSM right input multiplexers.
2. **PSM Right:** Horizontal nets, starting at PSM right output multiplexers and ending at PSM left input multiplexers.
3. **PSM Top:** Vertical nets, starting at PSM top output multiplexers and ending at PSM bottom input multiplexers.
4. **PSM Bottom:** Vertical nets, starting at PSM left output multiplexers and ending at PSM right input multiplexers.
5. **PSM Internal:** Internal nets within the PSM, realizing the Wilton switch pattern.
6. **CLB Input:** Nets starting at the output of the CBR and ending at the input of the LUT.
7. **CLB Output:** Nets starting at the LUT output and ending at the input of the CBW.

Our definition of uniformity then essentially measures differences between rows within a set, but no uniformity is guaranteed between the sets themselves. This definition is formalized in the following equations:

$$\mu_{c,n} = \frac{1}{N_R} \sum_{r=1}^{N_R} t_{c,r,n} \tag{1}$$

$$\sigma^2_{c,n} = \frac{1}{N_R} \sum_{r=1}^{N_R} \left(t_{c,r,n} - \mu_{c,n}\right)^2 \tag{2}$$

$$\overline{\sigma} = \frac{1}{N_C N_N} \sum_{c=1}^{N_C} \sum_{n=1}^{N_N} \sqrt{\sigma^2_{c,n}} \tag{3}$$

$$\overline{c_v} = \frac{1}{N_C N_N} \sum_{c=1}^{N_C} \sum_{n=1}^{N_N} \frac{\sqrt{\sigma^2_{c,n}}}{\mu_{c,n}} \tag{4}$$

Here, $t_{c,r,n}$ is the delay of a net in class n, column c and row r. Equation 1 provides the arithmetic mean $\mu_{c,n}$ of the delays, calculated over the N_R V-FPGA rows. $\sigma^2_{c,n}$ then calculates the variance for a net class in a certain column over the rows. This is further used in $\overline{\sigma}$ to calculate the arithmetic mean of the standard deviations of all net classes in all columns. $\overline{c_v}$ provides the arithmetic mean over the coefficient of variation of all net classes in all columns. Whereas the standard deviation is an absolute value and therefore depends on the mean of the delays, the coefficient of variation provides a relative measurement. As the delays in the host FPGA are largely discrete (e.g. fixed delays in LUTs), it is expected that relative delays can not be reduced further at some point. Because of this, we use $\overline{\sigma}$ to guide the design of our strategies and for evaluation of practically achievable uniformity. $\overline{c_v}$ is used to judge the quality of results for V-FPGA: As a smaller delay τ allows to put more logic elements in a path at the same frequency for V-FPGA applications, a constant standard deviation leads to reduced certainty of the number of V-FPGA logic elements in the path. A constant relative value $\overline{c_v}$ signifies unchanged conditions for the V-FPGA application synthesis.

3.3.2 Delay

Delay for the V-FPGA is a measurement that determines the final achievable V-FPGA user application frequency. Whereas the final frequency of user applications ultimately depends on that applications themseves, i.e. the length and nature of the critical path, the delays of individual elements within that path are determined by the placement of the V-FPGA onto the host-FPGA. For characterization and modelling of the V-FPGA architecture in VPR and for

application routing, the delays of the previously mentioned net classes are analyzed individually. But without knowledge of the final user application, none of the types can be considered as more important than any other. Therefore, which of the types ultimately will limit the user application frequency cannot be determined when implementing a generic V-FPGA. As a consequence, we reduce these various measurements to one metric for performance evaluation, the worst delay across all net classes. Equations to find the maximum delay are given below:

$$\tau_{c,n} = \max_{r \in \{1..N_R\}} t_{c,r,n} \tag{5}$$

$$\tau = \max_{c \in \{1..N_C\}} \max_{n \in \{1..N_N\}} \tau_{c,n} \tag{6}$$

Here, $\tau_{c,n}$ selects the worst delay of a net class (n) in a column (c), calculated over the V-FPGA rows. τ uses this to find the absolute maximum delay over all columns and all net classes in the design, providing the single value for evaluation.

3.3.3 LUT Overhead

Area is measured in number of host FPGA CLBs used by the V-FPGA design. Used CLBs in the design do not solely consist of the V-FPGA building blocks: It also includes CLBs that are constrained to be explicitly not used in placement, optimizing the placement regularity. For partition blocks, their size may need to be slightly more than the minimum required area, as aiming for utilization ratio of 100 % may cause routing to fail. 87 % utilization rate is the default target chosen by Vivado and is our starting point for manual placement. Results presented in [6] described this overall area, whereas the results in this publication will focus on the intrinsic LUT overhead. This overhead is caused by restrictions for the synthesis passes, caused by additional constraints used by the strategies. One example here is prevented LUT recombination.

3.4 Placement Methodology

Our overall approach to implement and evaluate the custom placement strategies for the V-FPGA consists of three steps: At first, the V-FPGA code is synthesized in Xilinx Vivado. Second, we run custom Tool Command Language (TCL) scripts on the synthesized design, adding various timing and location constraints. The third step is needed to evaluate the results by running a custom TCL script to extract timing and area information.

3.4.1 Synthesis

Synthesis largely follows the Vivado synthesis flow. To ensure proper conditions for the TCL script, some settings are adjusted: The *flatten_hierarchy* option is changed from the default *rebuilt* to *none*. As V-FPGA designs often provide customizable parameters [12], the default option *rebuilt* leads to unpredictable signal names when these parameters change. Changing this setting can also affect optimization across hierarchy levels. To limit the impact of this, the implemented design was analyzed manually and some optimization have been carried out manually in the VHDL source code.

3.4.2 Applying Manual Placement Strategies

After a design has been synthesized, we apply our custom placement strategies. The strategies will be described in detail in the next chapter, but all of them are based on the following constraints:

Timing Constraints As Vivado analyzes every possible path in the design, it will also consider configurations of PSM multiplexers that can create combinational loops. It is therefore not easily possible to constrain the timing of the design by simple definition of the final clock period, as Vivado will break the loops at arbitrary points. This generates long paths through different numbers of CLBs and PSMs, making it further impossible to constrain a path just between two specific PSMs. To solve this problem, these paths are broken manually.

We evaluate two variants of constraints used: In the variant with fine grained constraints, all individual atomic nets have their delay constrained using the *set_max_delay* timing exception, ensuring that the design still meets timing constraints and forcing the timing driven optimization to operate. These constraints will lead to path segmentation, which in this case is the desired outcome. In addition, it will add false path constraints on the original long paths automatically. Path segmentation can affect logic placement and timing results, so special care needs to be taken when examining the Vivado timing reports. We therefore use custom scripts to evaluate the delays of relevant nets instead. As will be shown in the result sections, these fine-grain constraints are necessary to animate Vivado to optimize the routing for the manually placed design. The drawback with this approach concerns scalability, as large designs which introduce many of these constraints cause excessive memory use and runtime in the Vivado toolflow. We therefore also evaluate variants without the fine-grain constraints, to determine whether they are really necessary.

In addition to path constraints, we define four clocks for our design: The primary clock as well as three auxiliary clocks used for the configuration of PSM, IOB and CLB elements. The frequency of configuration logic is less important than the application frequency, so configuration clocks will target a lower frequency, avoiding over-constraining the design. Moreover, we specify the relation between these clocks using an asynchronous clock group.

Placement Constraints Placement constraints are used to perform floorplanning by definition of pin placement and absolute, or relative, placement of cells. It guides and controls where the place-and-route tools may put FPGA design elements. Vivado supports various placement constraints, ranging from just constraining a group of logic in a certain area to exact placement of single cells to a certain logic element. We make use of the following placement constraints:

1. **LUTNM and HLUTNM:** Used to place two combinational functions into the same LUT.
2. **PROHIBIT:** When the only requirement is to avoid placing any logic at a specific site, this is achieved using this constraint.
3. **LOC and BEL:** To place a logical element in a specific location, we use the *place_cell* command. This command translates into LOC and BEL constraints, where LOC links the element from the netlist to a slice and BEL places it to a specific LUT or flip-flop within the slice.
4. **PBlock:** A PBlock is a collection of cells in one or more rectangular regions that specify the device resources contained by the block. It is more restrictive than no placement constraints, but less constraining than LOC and BEL.

3.4.3 Extracting Metrics

The Vivado timing report includes all details to judge in what respects the design met the timing constraints and usually provides the authoritative source in knowing the delay of all nets. But in case of the V-FPGA, this report can not be used to extract meaningful data: The combinational loops, path delay constraints and path segmentation hide the important delays of the atomic nets from the timing report. Even though the target value for these nets is given using the path delay constraints, it is still useful to extract the real delays. To remedy this, a TCL script was written to extract the delays manually, using the *get_net_delay* command to get the delays of atomic nets.

The process of calculating the delay through the CLB is divided into two parts. At first, we get the worst delay from the output of any multiplexer of the CBRs cells to the input of the LUT. Then we add the propagation delay from the output of the LUT to the CLB output, taking into consideration the maximum of the two paths of either bypassing the D flop or using it. The V-FPGA LUT can be implemented in three different ways by host-FPGA toolchain in one or two slices. All of these options are taken into account.

For the PSM net delays, the total delay from one PSM output to another PSM input is calculated. If such a net is divided into two or three parts due to interruption by IOB connection boxes, the parts are summed up. The delays of the horizontal tracks of the bottom border PSM are expected to be larger than the rest, as it is divided by the connection box of the upper tile and the IOB. In contrast, the horizontal tracks of the upper border PSM have the lowest delays because it is just interrupted by the IOB unit of the same tile. All delays are stored in a file for later evaluation grouped into the previously mentioned net classes.

Additionally, due to the structure of Xilinx 7 series LUT, more than one function can be implemented on the same fracturable LUT. This must be taken into consideration when determining the area used, or when deriving the minimum size needed for a tile. To find combined LUTs, we first get all the LUT BELs used as LUT6. We then get all the LUT5 in the design and compare their location with the LUT6, checking if they are located at the same site.

3.5 VPR Architecture Models

In order to synthesize user applications for the V-FPGA, we used the commonly used open-source VPR toolchain. For synthesis and placement, VPR needs an architecture description. This file is essentially a specially formatted XML file with information about the target FPGAs logical structure and timing information. We use a template for the logical structure, but timing information will be deduced from the extracted metrics for all analyzed design variations. To analyze effects of the various techniques on user applications, we make use of VPRs benchmark framework. As the benchmarks shipped with VPR are too large for the V-FPGA sizes compared here, we used 26 Verilog implementations of 74xx series ICs and one 4 bit full-adder implementation to emulate a longer critical path. We measure the maximum frequencies VPR achieves for those benchmarks, normalize them to the values obtained for the placement based on Vivado standard placement and lastly average the results for all benchmarks. Averaging over those circuits furthermore provides a certain immunity against non-stable results, which are caused by pseudo random start conditions in placement algorithms.

4 Logic Placement Strategies

In the following, we discuss the three manual placement strategies in detail. We primarily use the uniformity metric to guide development of the strategies, then assess critical path delay and LUT overhead in the evaluation.

4.1 Standard Vivado Placement

Figure 3 shows placement result of the default Vivado strategy (Vivado Synthesis Defaults, Vivado Implementation Defaults, Vivado 2019.1.1). The figure illustrates the arguments given previously in the introduction section: Automated placement does not make explicit use of the structural regularity of the V-FPGA, which results in tiles being implemented in slightly different ways. Some are more distributed, others more localized, leading to varying net delays and reducing uniformity of the V-FPGA architecture. This effect is even more apparent in larger designs, where placement algorithms have to deal with an overall larger amount of nets and cells. These standard Vivado placement results will be used as reference point for the evaluation of the manual placement strategies presented here. Evaluation results will be normalized accordingly to show improvement or degradation over the usual automated approach.

4.2 General Custom Approach

Before the strategy-specific placement step, some steps common to all approaches are required.

4.2.1 V-FPGA Size Determination

Before the V-FPGA can be placed, a suitable host FPGA location and area has to be determined. This step has to consider non-uniformity of the host FPGA: As described previously,

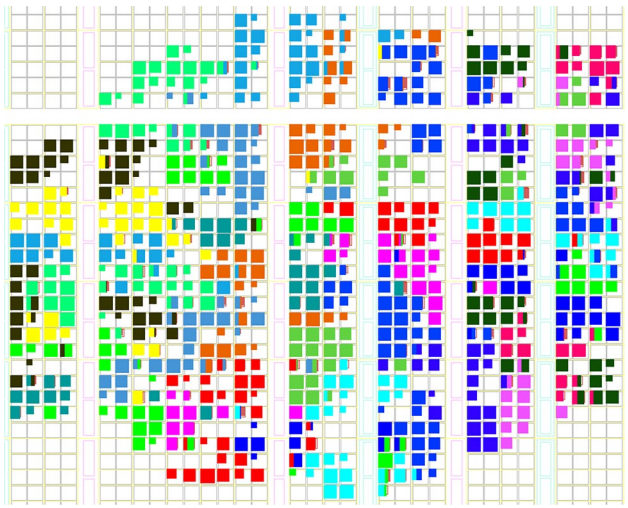


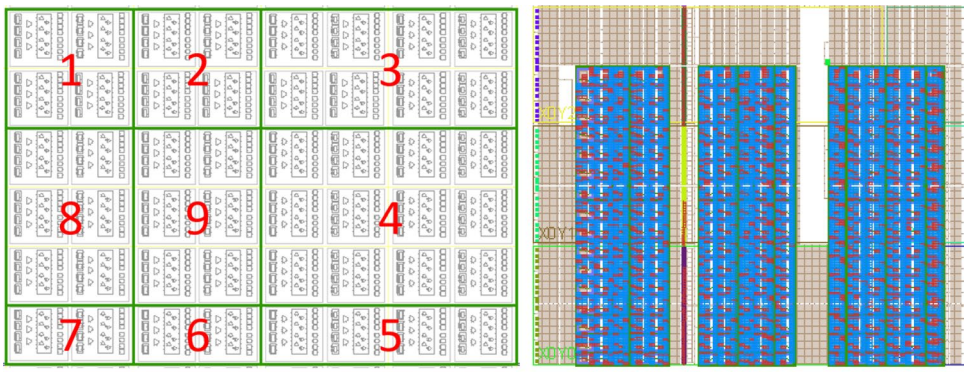
Figure 3 V-FPGA placed using standard Vivado placement. Host-FPGA CLBs belonging to same V-FPGA tile are shown in the same color. As can be seen, some tiles are compact, whereas some are scattered across wider area. It can also be seen that all-in-all, Vivado packs tightly and does not keep empty sites to preserve overall structure.

Virtex 7 devices have a rectangular structure with larger gaps (more than 2 columns) and smaller gaps (2 columns) between CLBs, caused by I/O banks, clocking and other support logic. In addition, DSP and BRAM blocks are distributed over the chip between CLB columns. In order to reduce net length between placed V-FPGA logic blocks, the biggest area with no large gaps will be selected, where the threshold when a gap is considered large is configurable. This is supposed to improve net delays and support the rectangular layout of the V-FPGA. Finding the location and area consists of the following steps:

1. Estimate the overall area needed for the design using total CLB count.
2. Create a 2D array which represents available and used CLBs. Then search for the largest possible target area, only considering areas without blockages larger than the accepted gap threshold. A reasonable value for Virtex 7 is two, allowing DSP and BRAM gaps but avoiding larger ones.
3. Calculate tile dimensions, fitting all tiles in square form in the target area.
4. If the previous step fails, set the dimension ratio of all tiles relative to the vertical and horizontal dimensions of the selected target area.
5. If the largest contiguous target area is not large enough to fit the complete design (step three and four failed), the steps are reevaluated. In this reevaluation, multiple disconnected areas are allowed, yielding split target areas as shown in Fig. 4b.
6. Find the vertical and horizontal dimensions of all tiles according to their resource usage.
7. Normalize the dimensions of all tiles in the same column or row.

4.2.2 Rectangular and Quadratic Tiles

Before a tile can be created, its size must be determined. We investigate two options to derive the tile size: The first option is to use a size with same width and height for all tiles, resulting in quadratic tiles. The largest tile dimensions are then taken as the unified size for the PBlocks of all tiles. Alternatively, the size can be chosen according to the required area in each tile. This requires additional rules for tile sizes, to keep the rectangular layout of V-FPGA and avoid irregular layout results. Therefore, the horizontal size for all tiles in the same column and the vertical size for all tiles in the same row have to be identical, leading to rectangular tiles. Figure 4a demonstrates the second option, showing the generated floorplan for a small 2x2 CLB V-FPGA. For all strategies introduced, we will evaluate the variant with rectangular and with quadratic tiles. We will assess differences between these options in the evaluation section.



(a) Simple Floorplan. **(b)** Split Target Area.

Figure 4 Floorplanning and PBlocks: **a**: Floorplan for 2x2 CLB V-FPGA using individually calculated sizes for each tile. To keep the layout regular, widths and heights of tiles are adjusted accordingly. **b**: PBlock floor plan demonstrating a larger V-FPGA design. The start point location was forced, so the target area had to be split into three smaller areas because of intersecting hard blocks.

4.2.3 Split Area Implementation

If the V-FPGA does not fit into the largest available contiguous area when considering large gaps according to the threshold, our placement script can split the target area into multiple sub-areas, making sure not to split V-FPGA tiles across sub-areas. In such cases (happening for very large V-FPGA designs), uniformity will be degraded due to the nets crossing gaps between the sub-areas. The placement script will ensure not to split a V-FPGA tile into multiple sub-areas, to at least guarantee better uniformity within the tile in such cases.

4.3 Basic PBlock Strategy

In the basic PBlock strategy, we contain each tile in a single Partition Block (PBlock): We create a block with suitable size in quadratic or rectangular form, then use the *add_cells_to_pblock* TCL command to add all cells of a tile to the block. After the tile size has been determined, the host FPGA location and target area will be fixed. Finally, all V-FPGA cells are fixed to the PBlocks belonging to their tile using the PBlock constraints, completing the basic PBlock placement. Exemplary placement results are shown in Fig. 4.

4.4 Nested PBlock Strategy

In addition to the PBlocks used in the first strategy, this strategy introduces up to two additional PBlocks within each tile. Logic belonging to the V-FPGA CLBs and IOBs is mapped to these nested PBlocks accordingly: When defining the PBlocks, all assigned logic cells are forced into the blocks, but this does not prevent placing any additional unassigned cells into them. Based on this idea, we introduce two more variants in addition to the rectangular vs. quadratic layout distinction: In the partially nested strategy, we use the outer PBlock for the tile and nested blocks for IOB and CLB, but the PSM is only constrained by the outer PBlock. This gives Vivado the freedom to place the PSM in the remaining outer PBlock area, or place part of it inside the nested PBlocks. In the fully nested strategy, we force Vivado to not place any PSM logic in the nested PBlocks, prohibiting usage of remaining logic cells in them. Figure 5 demonstrates the concept for a 5x5 CLB V-FPGA.

The placement script is extended with the following steps to create the nested PBlocks:

1. The internal PBlocks can consist of multiple rectangles. The CLB PBlock is placed at the bottom left corner with height at most equal to the height of the tile minus

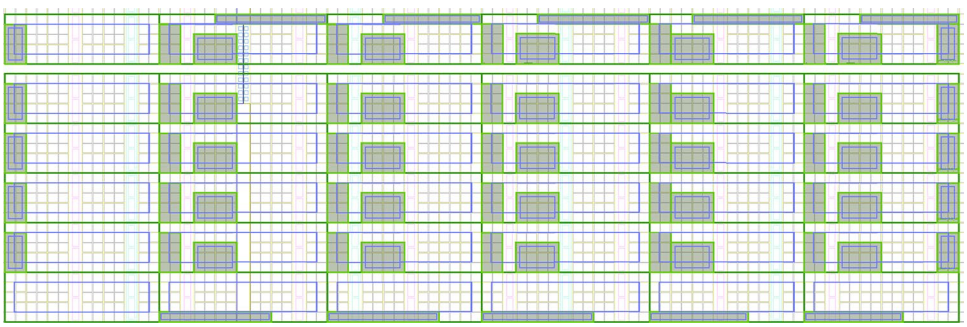


Figure 5 5x5 CLB V-FPGA floorplan with nested PBlocks. The nested CLB PBlock is divided into two pieces to ensure the minimum possible area is used. The top right corner tile has an extra nested PBlock for its second IOB unit. No internal PBlock was used at all in the bottom left corner tile, as it only contains a PSM.

one. This guarantees some freedom to IOB PBlock and to ensures distribution of the PSM unit over the tile PBlock.
2. The IOB PBlock is placed within the tile PBlock. The side is determined according to the tile type.

4.5 Fine-Grain Manual Placement Strategy

This strategy further constrains logic, directly mapping the relevant LUTs and flipflops to specific LUTs or flipflops in the 7 series host CLB. As there are numerous ways to place the logic within a tile, a manually derived layout is chosen instead of trying to find a fully automated one. The strategy is then made generic to support different V-FPGA parameters, but the layout is fixed to the V-FPGA and therefore cannot be reused for completely different applications. Evaluation of different manual layouts led to a placement as was presented in Fig. 1a: The PSM is located in the upper right corner and the CLB is placed in the lower part of the tile. Figure 6 shows the device view in Vivado after the manual placement strategy has been applied.

The implementation of this strategy operates on two lists for each tile PBlock, an instruction list and a list of the free host FPGA LUTs. The instruction list contains simple V-FPGA logic element place instructions, interleaved with sorting instructions. It is processed element by element, either placing logic elements or resorting the list of free resources. When an element placement instruction is processed, the logic elements are mapped sequentially to the elements in the sorted list of free resources, starting at a specified offset. When a resorting instruction is found, the resorting algorithm sorts the list of remaining available host LUTs. It sorts horizontally or vertically and uses ascending or descending sorting order, depending on the instruction. As an example, the *sort_xy_dd* instruction sorts first based on the x location, and if the x value is the same for some CLBs, it uses y as secondary criteria. Descending sorting is applied in both cases. This specific instruction is used to sort the list of available logic elements before placing the right and left multiplexers of the PSM, as they need to be placed vertically from the top right corner. Sorting is always done on the list of free resources, so the length of this list decreases as the placement process proceeds. This makes it possible to reach every single CLB in the PBlock, not just the ones at the borders.

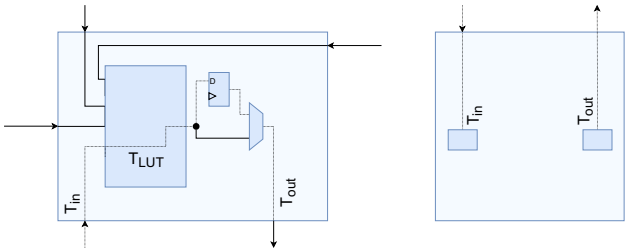


Figure 7 VPR architecture modeling the V-FPGA structure with the CLB model on the left-hand side and the IOB model on the right-hand side. Dotted lines show timing paths which need to be characterized and specified to enable user application synthesis with VPR.

5 VPR Architecture Generation

To describe the V-FPGA architecture and timings for VPR, an architecture XML file has to be provided. The logical architecture description for the V-FPGA consists of CLBs and IOBs and closely resembles the V-FPGA VHDL implementation as it was previously described. The most important aspect of the VPR architecture model is the timing information, which describes delays for certain paths. Figure 7 shows the simplified CLB block and IOB block as modelled and depicts some examples of timing paths. For the IOBs, we directly specify input delays and output delays. Only one of those is relevant in each single IOB, depending on the

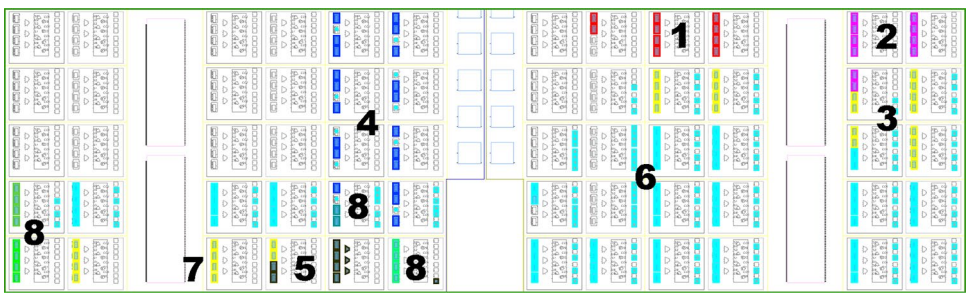


Figure 6 V-FPGA tile (type 9) placed using the manual placement strategy. Multiplexers of the PSM's top, right, bottom and left side are marked red (**1**), purple (**2**), yellow (**3**) and blue (**4**). The CLB is located at the bottom with the LUT, two internal multiplexers and D-flipflop colored in black (**5**). Sky blue color represents the configuration units of the tile (**6**). Yellow blocks at the bottom (**7**) depict Write Connection Boxs, whereas Read Connection Boxs are marked green and turquoise (**8**) and make up remaining logic distributed around the LUT.

mode (input or output) in which it is used. The timings are directly obtained from the placed V-FPGA design using the timing extractions scripts. For the simplified CLBs model, three timing delay values are required: First, the input delay T_{in}, which models the delay through the CBR and input multiplexer crossbar to the LUT. Second, the LUT delay itself. As the V-FPGA LUT maps directly to a host FPGA LUT, this value is the propagation delay though the host FPGA LUT and can be obtained from the host FPGA datasheet. The third value, T_{out}, is the output delay from LUT to the output of the CBW. Here two paths could be considered independently: The one starting at the D-FF and passing through the multiplexer, when the LUT output is registered. The other one starting at the LUT output and passing through the multiplexer directly, for combinational outputs. To simplify modeling, we use the maximum of those values for both paths. The simple V-FPGA architecture does not have any CLB internal feedback path, so no additional delay values have to be considered for the CLB. For the PSM, the VPR model description is unfortunately not very flexible. It is possible to assign resistance and capacitance to wires and to assign constant delays to the switches in PSMs. But most notably, it does not seem possible with a normal architecture description to actually assign different delays to individual PSMs.

To obtain customized architecture descriptions for the various tested implementation strategies and V-FPGA sizes, we modelled the base architecture as a parametrizable template. The template is essentially a mustache file, which can then be combined with the timing descriptions in form of a simple JSON file to yield the architecture with custom timings. The JSON file itself is obtained through evaluation of the detailed timing information generated by the TCL timing extraction scripts. In some cases, the paths depicted in Fig. 7 consist of multiple host FPGA segments or logic elements, so those elements are summed in the timing extraction scripts. We then consider two main approaches: The basic, direct modeling of CLB, IOB and PSM. In this case, we check all CLBs, IOBs or PSMs in the V-FPGA and search for the ones with the maximum delay. We then use this value for all logic elements of that type within the VPR architecture. This clearly reduces overall performance by assigning worse timings to all but the slowest logic blocks, but it is the approach closest to traditional FPGA architecture modeling, where uniform delays are expected and common. It therefore allows VPR to work efficiently in the way it was originally meant to be used.

The second approach tries to achieve better results by modelling each CLB individually. Here, a different block type is modeled for each single CLB in the architecture. Timings are extracted for each individual V-FPGA CLB and assigned to the block at that specific location in the V-FPGA architecture description. Making use of the template approach, automatization of this approach is rather simple. Nevertheless it results in large FPGA architecture files and causes certain compute and memory overhead in the VPR tools. Whereas a similar approach for IOBs and PSM could further increase performance, there are reasons against individual characterization of those: For IOBs, the expected gain is assumed too be small considering the VPR runtime increase for common designs, where IOB paths are rare compared to logic routing between CLBs. For PSMs, this approach is not feasible, as VPR enforces uniformity of the interconnect network.

6 Evaluation of the Placement Strategies

To evaluate the different placement strategies with different V-FPGA parameters, three V-FPGA designs of increasing size (2x2, 5x5 and 8x8) have been implemented: Unlike in [6], we kept the channel width value at 4 for all architectures. Exact numbers therefore are different than in [6], but this allows for more stringent comparison between different V-FPGA sizes, avoiding varying routing congestion effects in the different designs. All designs have been evaluated both with fine grain timing constraints and without fine grain timing constraints. We compare the three strategies presented previously, with both quadratic and rectangular tiles for the PBlock strategies and two different placement script variations for fully manual placement. Comparisons between the three proposed strategies are held in the previously described metrics of uniformity, worst delay and LUT overhead, where results are normalized to the standard Vivado strategy.

For the remaining discussion, we introduce the following abbreviations: *T* in a design evaluation means that we used fine grain timing constraints for this design, e.g. *2x2 T* means the 2x2 design with fine grain timing constraints applied. The implementation strategies are abbreviated *I1a* to *I4b* and refer to the following strategies:

I1a: Basic strategy using quadratic PBlocks.

I1b: Basic strategy using rectangular PBlocks.

I2a: Partially nested strategy using quadratic PBlocks.

I2b: Partially nested strategy using rectangular PBlocks.

I3a: Fully nested strategy using quadratic PBlocks.

I3b: Fully nested strategy using rectangular PBlocks.

I4a: Fully manual placement strategy, using sort instruction list a.

I4b: Fully manual placement strategy, using sort instruction list b.

Table 1 Normalized standard deviation $\overline{\sigma}$ for synthesis strategies in Vivado.

Design	S1	S2	S3	S4	S5	S6	S7	S8
2x2	1.00	**0.92**	**0.92**	1.00	1.20	**0.92**	**0.92**	1.01
5x5	1.00	**0.91**	**0.91**	1.00	1.13	1.02	1.02	0.99
8x8	1.00	**0.94**	**0.94**	1.00	1.16	1.00	1.00	1.00
2x2 T	1.00	1.02	1.02	1.00	1.32	**0.87**	**0.87**	1.31
5x5 T	**1.00**	1.04	1.04	**1.00**	1.53	1.14	1.14	1.08
8x8 T	1.00	1.00	1.00	1.00	1.19	**0.85**	**0.85**	1.02

Bold values denote the best result for a given design

6.1 Impact of Synthesis Strategies

In addition to the evaluation of the standard synthesis strategy, we also tested the various other synthesis strategies offered by Vivado, assessing their impact on the overall results. In the following, the available synthesis strategies will be abbreviated as follows:

S1: Vivado Synthesis Defaults

S2: Flow_AreaOptimized_high

S3: Flow_AreaOptimized_medium

S4: Flow_AreaMultThresholdDSP

S5: Flow_AlternateRoutability

S6: Flow_PerfOptimized_high

S7: Flow_PerfThresholdCarry

S8: Flow_RuntimeOptimized

Tables 1 and 2 show the uniformity metrics, where best results for each design are marked in bold text. The $\overline{\sigma}$ and $\overline{c_v}$ values show similar trends in overall, which is expected as those are closely related metrics. Of the tested strategies, no strategy shows consistent improvements in uniformity for all design sizes. Strategies S2 and S3 yield improvements when not using timing constraints, but they yield worse uniformity for some cases when using timing constraints. Strategies S6 and S7 yield uniformity improvements for the 2x2 and 8x8 designs with timing constraints, but do yield worse results on the 5x5 design. On the 5x5 design with timing constraints, all synthesis strategies yield worse results than the default strategy. It is therefore not possible to choose a single synthesis strategy which yields best uniformity for all design sizes. This is not entirely unexpected, as non of the synthesis strategies is optimized for uniformity.

The maximum delay as shown in Table 3 is larger for the non-default synthesis strategies in almost all cases. S6 and S7 yield better results when the designs are not timing constrained and S5 yields good results in two timing constrained cases. Further analysis shows that those are however exceptional cases and the results depend a lot on channel width, the V-FPGA size and other structure parameters. These synthesis strategies can therefore not be recommended for all cases: If an improvement in uniformity or delay is wanted, those strategies need to be evaluated for the specific use case, which makes consistent improvements using manual placement even more important.

6.2 Uniformity

Tables 4 and 5 show uniformity metrics for the various manual placement strategies. Again patterns for $\overline{\sigma}$ and $\overline{c_v}$ values are similar, as those are closely related. We expect the rectangular tile versions to perform slightly better, as they reduce wire length between the tiles. when comparing the *b* variants to the *a* variants, this effect can be seen in the table, although it is subtle. We further expected the fully manual strategies *I4a* and *I4b* to yield best uniformity, as they enforce most constraints on the design. This

Table 2 Normalized variation $\overline{c_v}$ for synthesis strategies in Vivado.

Design	S1	S2	S3	S4	S5	S6	S7	S8
2x2	1.00	**0.90**	**0.90**	1.00	1.23	1.01	1.01	1.02
5x5	1.00	**0.96**	**0.96**	1.00	1.19	1.08	1.08	0.99
8x8	1.00	**0.97**	**0.97**	1.00	1.18	1.05	1.05	1.00
2x2 T	1.00	0.98	0.98	1.00	1.14	**0.90**	**0.90**	1.10
5x5 T	**1.00**	1.05	1.05	**1.00**	1.43	1.13	1.13	1.04
8x8 T	1.00	0.94	0.94	1.00	1.06	**0.86**	**0.86**	0.95

Bold values denote the best result for a given design

Table 3 Normalized worst delay τ for different synthesis strategies in Vivado.

Design	S1	S2	S3	S4	S5	S6	S7	S8
2x2	1.00	1.25	1.25	1.00	1.07	**0.96**	**0.96**	1.00
5x5	1.00	0.94	0.94	1.00	1.02	**0.94**	**0.94**	1.00
8x8	1.00	1.08	1.08	1.00	1.23	**0.88**	**0.88**	1.00
2x2 T	1.00	0.96	0.96	1.00	**0.94**	0.97	0.97	1.09
5x5 T	**1.00**	1.11	1.11	1.00	1.12	1.01	1.01	1.12
8x8 T	1.00	1.13	1.13	1.00	**0.95**	1.00	1.00	1.29

Bold values denote the best result for a given design

Table 4 Standard deviation $\overline{\sigma}$ relative to Vivado standard implementation results, comparing uniformity.

Design	I1a	I1b	I2a	I2b	I3a	I3b	I4a	I4b
2x2	0.93	0.80	1.00	0.97	**0.78**	0.90	**0.78**	0.87
5x5	0.84	0.83	0.84	0.82	0.93	0.81	0.80	**0.77**
8x8	0.89	0.86	0.88	0.84	0.89	**0.81**	**0.81**	0.84
2x2 T	0.96	0.94	0.93	**0.86**	1.11	1.01	0.93	0.92
5x5 T	0.92	0.95	0.95	0.93	0.95	0.86	**0.82**	0.87
8x8 T	0.82	0.83	0.81	**0.76**	0.83	0.80	0.79	0.80

Bold values denote the best result for a given design

is however not always the case, for small designs some of the other strategies yield better results. Whereas the best strategy varies depending on the design, most strategies yield more uniform results than the standard placement strategy in all cases. The fully manual strategy *I4a* for example yields consistent results for all architectures and when using timing constraints, it improves for larger designs. This is expected and less an improvement of the manual strategy than a degradation of the default strategy for larger designs. As designs become larger, routing requires more effort, ultimately reducing the quality of results. In addition, it should be noted that designs with timing constraints are more uniform than those without in general, but this is not visible in the tables as the strategies are normalized to their relative default strategy with same constraint application.

6.3 Delay

Figure 8 shows maximum delay for the various different strategies evaluated. Overall delay is largely dominated by the routing. As the routing is still performed by Vivado's automated router, timing constraints directly influence the optimization effort. Due to this, most strategies yield worse delays when no constraints are given. Because of the reduced packing density especially in small designs, total delays may be larger than in the standard placement. The fully manual strategies *I4a* and *I4b* perform well even in those cases, as the LUT location constraints leave less decisions to Vivado than PBlock based placement. For the timing constrained designs, most strategies lead to improvements. Those are not as large as initially reported in [6], which is mainly caused by the fixed and smaller V-FPGA track width in this work compared to the previously reported results. This track width of 4 is also the reason for worse results in the 2x2 cases, as those were evaluated with channel width 2 in [6]. The *I4a* strategy again is most stable across design sizes, which makes it a candidate to be a default strategy.

6.4 LUT overhead

Our previous publication [6] showed an increased area usage of up to 16 % when considering the total V-FPGA size, e.g.

Table 5 Coefficient of variation $\overline{c_v}$ relative to Vivado standard implementation results, comparing uniformity.

Design	I1a	I1b	I2a	I2b	I3a	I3b	I4a	I4b
2x2	0.93	0.85	0.92	0.90	**0.75**	0.86	0.86	0.97
5x5	0.87	0.89	**0.84**	0.85	0.93	0.82	0.91	0.94
8x8	0.90	0.89	0.89	0.86	0.90	**0.80**	0.93	0.98
2x2 T	0.86	0.85	0.83	**0.79**	0.90	0.87	0.87	0.91
5x5 T	0.93	0.95	0.92	0.93	0.90	0.87	**0.86**	0.95
8x8 T	0.83	0.85	0.83	**0.77**	0.84	0.81	0.84	0.87

Bold values denote the best result for a given design

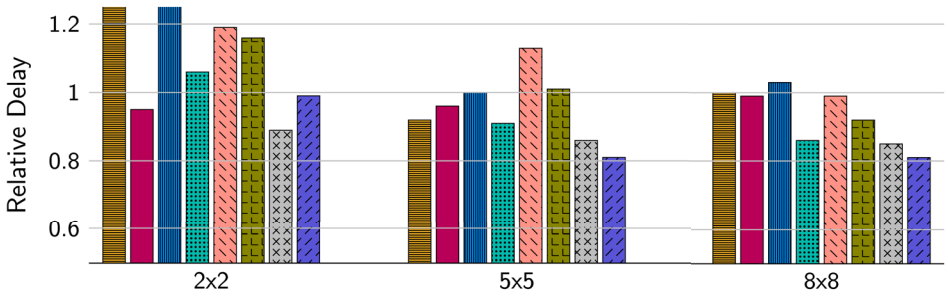
counting unused CLBs in the area of the V-FPGA as part of the used CLBs. This overhead is expected, as the target utilization rate for the are was set as 87 % in the placement scripts. In addition, we evaluated the overhead of the strategies in raw LUT numbers: Due to prevented LUT recombination and similar effects, we do expect to see some differences in the amount of LUTs which are required. As there was an maximum overhead of 1 %, we do not list these overheads for all strategies individually. We conclude that the constraints the strategies use do not have any meaningful impact on LUT count, the overall impact due to unused parts of P-Blocks and utilization rate is more significant and the main determining factor for area overhead.

6.5 VPR Architecture Performance

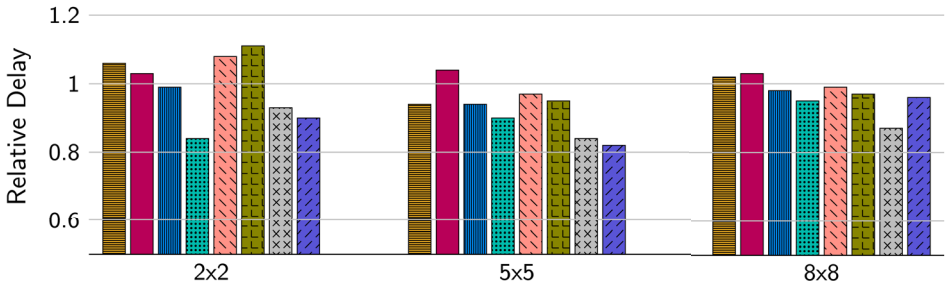
Figure 9 shows evaluations of 72 different VPR architectures, grouped first into whether fine grain timing constraints were applied (a and b) or not (c and d). Figures are further grouped according to the variant used to characterize the timing in the VPR architecture: Whether the worst case delays of all CLB has been used (a and c) or whether each CLB has been modeled individually (b and d). The subgraphs then are further divided into two groups according to the tested V-FPGA sizes. The 2x2 V-FPGA has not been evaluated, as it is too small for the benchmark circuits. Each set of results compares the default placement strategy and the manual placement strategies. Please note that all values in (a) and (b) are normalized to the default placement strategy with worst case timing characterization in (a), whereas the values in (c) and (d) are normalized to the default placement strategy with worst case timing characterization in (c). This is consistent with the way the previous results have been normalized and allows a direct comparison between individual timing characterization and the wort-case characterization. For example, comparing the standard placement of the 8x8 architecture in (b) to the reference in (a) shows that the individual characterization lead to 16 % faster clock frequencies in average for the evaluated benchmarks. Differences between the timing constrained and non-timing constrained variants are not directly visible in the graphs, but timing constrained designs were 46 % faster in average for the reference architectures.

When considering the achieved synthesis results for V-FPGA user applications, we can therefore reaffirm some of the conclusions based on uniformity and delay metrics in the previous section and add some additional ones: The evaluation of the benchmarks and the improvement of 46 % in clock frequency clearly shows that fine-grain timing constraints are

Figure 9 Average over achieved maximum frequencies in VPR benchmarks. Results are normalized relative to the standard synthesis strategy with worst case timing characterization. Timing constraint application of the reference strategy is the same as in the result that is normalized. Strategies shown from left-to right: Standard placement, I1a, I1b, I2a, I2b, I3a, I3b, I4a, I4b.

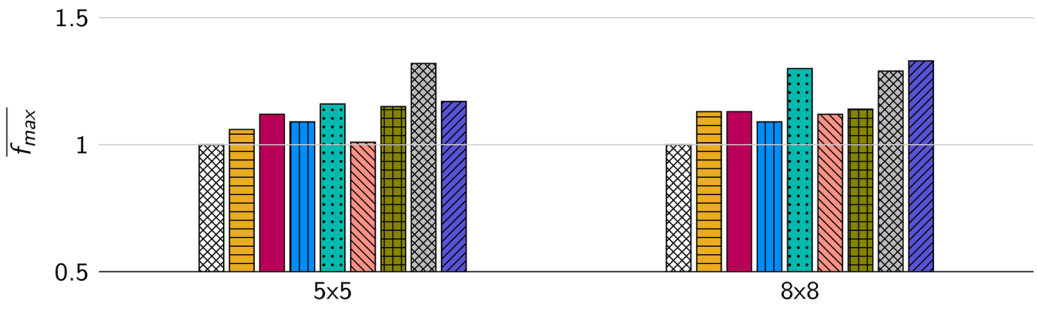


(a) Without fine grain timing constraints.

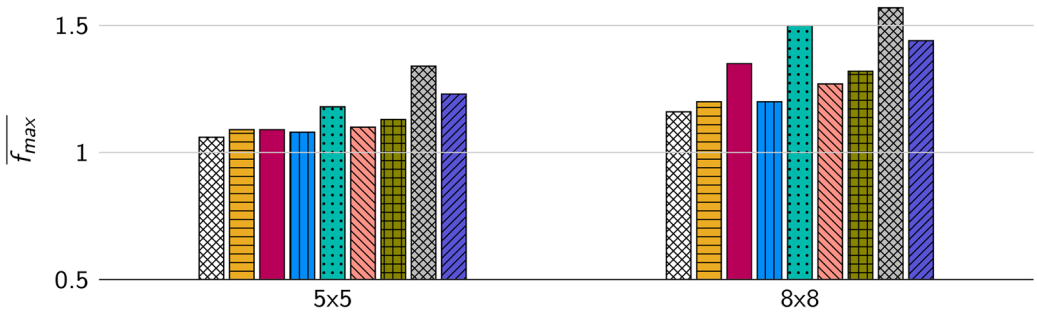


(b) With fine grain timing constraints.

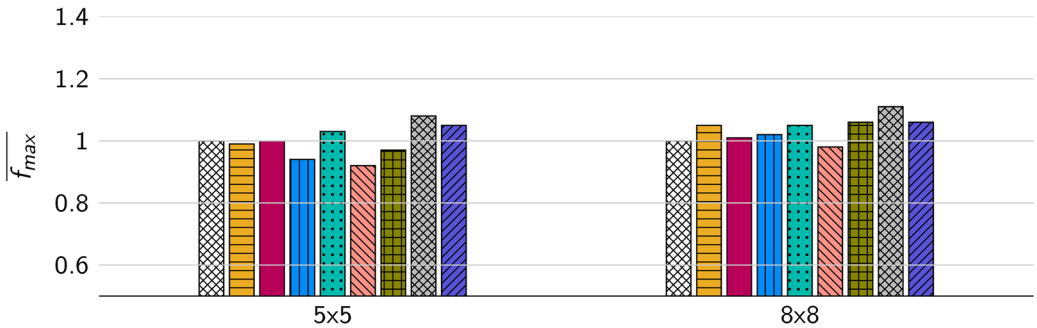
Figure 8 Delays τ relative to Vivado standard implementation results. Of all analyzed atomic nets, the largest increase or the smallest decrease, i.e. the worst case, is shown. Strategies shown from left-to right: I1a, I1b, I2a, I2b, I3a, I3b, I4a, I4b.



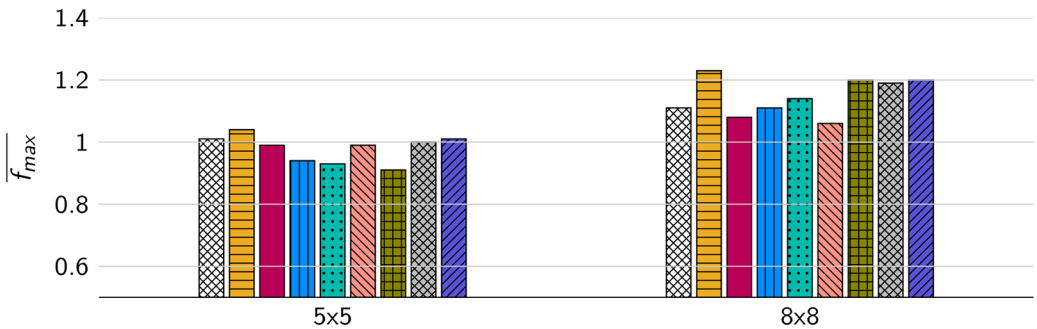
(a) No fine grain timing constraints, worst case timing characterization.



(b) No fine grain timing constraints, individual timing characterization.



(c) With fine grain timing constraints, worst case timing characterization.



(d) With fine grain timing constraints, individual timing characterization.

essential, especially for the default placement strategy. For the other strategies, there are still benefits of using timing constraints, but they are slightly reduced in comparison. The reason for this is that the manual placement strategies provide larger relative improvements for the non-timing constrained case. This can be explained by the manual strategies enforcing a uniform placement and a certain fixed layout, even without the timing constraints. The constraints still further improve the results for manual placement strategies, as they affect the routing in the global interconnect. For the default placement strategy and unlike for the manual placement strategy, they however also directly influence the placement and therefore have larger impact. It can further be seen that the individual characterization approach in VPR yields improvements of up to 28 % for the 8x8 designs without timing constraints. For the timing constrained designs, the improvements are slightly reduced at up to 14 %. This can be explained by an overall larger uniformity of placed architectures in the timing constrained case. As more uniformity means more CLBs are closer to the worst case timing, the benefit of modelling them individually compared to using the worst case is diminished.

Looking at Figures (c) and (d), another conclusion that can be drawn is that the simple P-Block based techniques I1a to I3b do not yield much improvements or even reduce performance compared to the standard placement for 5x5. Investigation shows that the low area utilization targeted caused logic to be spread more widely than in the default placement, which manages to pack all elements more closely. This increases routing delays, therefore making these strategies less suitable for small designs. The I4a and I4b strategies perform similar to the default strategy, do however not yield any improvements. Interestingly for the 5x5 designs, some strategies perform better in the simple characterization then in the individual timing characterization. This is unexpected, as the modeled individual architecture has strictly lower delay than the maximum one. Using exactly the same placement and routing as in the simple architecture should therefore yield higher frequencies. Investigation into the placed benchmarks shows localized higher routing channel utilization in the individual characterization. We conclude that VPR can not always efficiently handle an architecture the way we modeled it and that changes to the used algorithms may be required. For the 8x8 designs, the individual characterization yields better results than the simple characterization. Here, the increased nonuniformity of the architecture makes the approach more viable.

7 Conclusion

In this publication, we investigated various ways to increase the user application frequencies for V-FPGAs. We introduced a uniformity metric, arguing that the common basic timing modeling is based on the maximum delay in a design and therefore sensitive to outliers. After describing our toolflow to measure the delays and calculate the uniformity metric for our V-FPGA designs, we evaluated different Vivado synthesis strategies using that framework. Results show that no single synthesis strategy consistently provides better uniformity than the default synthesis. As another solution, we described three different manual placement approaches and evaluated the conditions under which those achieve more uniform results. Of those, strategy *I4a* showed consistently better results. To verify whether these results also enable higher user application frequencies as expected, we used the extracted metrics to model the architecture timings in VPR. Using a set of small benchmarks, we determined the maximum frequencies VPR could obtain for those benchmarks in our characterized architectures. Here depending on the architecture and whether fine-grain timing constraints were used, improvements of up to 33 % could be seen. For timing constrained designs, which perform better in general, only an improvement of 11 % could be achieved. Aiming to improve those results, we introduced a timing modeling approach for VPR which models all CLBs individually. Here improvements of up to 11 % were achieved for Vivado default placement and up to 14 % for manual placement strategies.

These results will allow for higher clock rates in user applications on V-FPGA, lowering the entry barrier for using it in more cases. In addition, the increased uniformity can be used to enable new concepts, such as overclocking of user applications or moving parts of the application dynamically on the V-FPGA. In general, more uniform V-FPGAs also provide more realistic insights for physical FPGA development, as they resemble their uniformity more closely. Future work in this area may focus on more advanced placement strategies, extending VPR to better handle individual timing in architectures and making the whole process more efficient and robust using tools such as RapidWright.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Peter Wagih and Johannes Pfau. The first draft of the manuscript was written by Johannes Pfau and Peter Wagih and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was supported by the German Research Foundation (DFG) within the PARFAIT project (DFG 326384402).

Availability of Data and Material Not applicable.

Code Availability The evaluation only used the industry standard program Xilinx Vivado which is a commercial program, but widely available. The custom TCL scripts developed for this research are tailored to and partially derived from the V-FPGA sources. V-FPGA sources have been previously been developed as part of research projects by

various authors. As licensing and copyright has not been cleared, the sources are unfortunately currently not publicly available. They are archived according to good scientific practice and DFG rules.

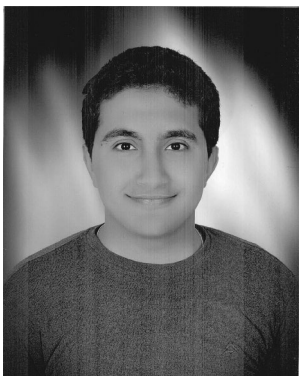
Declarations

Conflicts of Interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

1. Figuli, P., Hübner, M., Girardey, R., Bapp, F., Bruckschlögl, T., Thoma, F., Henkel, J., & Becker, J. (2011). A heterogeneous SoC architecture with embedded virtual FPGA cores and runtime Core Fusion. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)* (pp. 96–103). https://doi.org/10.1109/AHS.2011.5963922
2. Sidiropoulos, H., Figuli, P., Siozios, K., Soudris, D., & Becker, J. (2013). A platform-independent runtime methodology for mapping multiple applications onto FPGAs through resource virtualization. In *2013 23rd International Conference on Field programmable Logic and Applications* (pp. 1–4). https://doi.org/10.1109/FPL.2013.6645564
3. Harbaum, T., Schade, C., Damschen, M., Tradowsky, C., Bauer, L., Henkel, J., & Becker, J. (2017). Auto-SI: An adaptive reconfigurable processor with run-time loop detection and acceleration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)* (pp. 153–158). https://doi.org/10.1109/SOCC.2017.8226027
4. Pfau, J., Reuter, M., Hofmann, K., & Becker, J. (2020). Designing universal logic module FPGA architectures for use with ambipolar transistor technology. In *2020 International Conference on Field-Programmable Technology (ICFPT)* (pp. 165–173). https://doi.org/10.1109/ICFPT51103.2020.00031
5. Rai, S., Nath, P., Rupani, A., Vishvakarma, S. K., & Kumar, A. (2021). A survey of fpga logic cell designs in the light of emerging technologies. *IEEE Access, 9,* 91564–91574. https://doi.org/10.1109/ACCESS.2021.3092167
6. Pfau, J., Zaki, P. W., & Becker, J. (2021). Evaluation of different manual placement strategies to ensure uniformity of the V-FPGA. In S. Derrien, F. Hannig, P. C. Diniz, & D. Chillet (Eds.), *Applied Reconfigurable Computing. Architectures, Tools, and Applications* (pp. 35–49). Springer International Publishing, Cham.
7. Luu, J., Goeders, J., Wainberg, M., Somerville, A., Yu, T., Nasartschuk, K., Nasr, M., Wang, S., Liu, T., Ahmed, N., Kent, K. B., Anderson, J., Rose, J., & Betz, V. (2014). Vtr 7.0. *ACM Transactions on Reconfigurable Technology and Systems, 7*(2), 1–30. https://doi.org/10.1145/2617593
8. Yuan, J., Chen, J., Wang, L., Zhou, X., Xia, Y., & Hu, J. (2019). Arbsa: Adaptive range-based simulated annealing for FPGA placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38*(12), 2330–2342. https://doi.org/10.1109/TCAD.2018.2878180
9. Vansteenkiste, E., Kaviani, A., & Fraisse, H. (2015). Analyzing the divide between FPGA academic and commercial results. In *2015 International Conference on Field Programmable Technology (FPT)* (pp. 96–103). https://doi.org/10.1109/FPT.2015.7393137
10. Borriello, G., Ebeling, C., Hauck, S. A., & Burns, S. (1995). The triptych FPGA architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 3*(4), 491–501. https://doi.org/10.1109/92.475968
11. Shi, M., Bermak, A., Chandrasekaran, S., & Amira, A. (2006). An efficient FPGA implementation of gaussian mixture models-based classifier using distributed arithmetic. In *2006 13th IEEE International Conference on Electronics, Circuits and Systems* (pp. 1276–1279). https://doi.org/10.1109/ICECS.2006.379695
12. Figuli, P., Ding, W., Figuli, S., Siozios, K., Soudris, D., & Becker, J. (2017). Parameter sensitivity in virtual FPGA Architectures. In S. Wong, A. C. Beck, K. Bertels, & L. Carro (Eds.), *Applied Reconfigurable Computing. ARC 2017. Lecture Notes in Computer Science* (Vol. 1026, pp. 141–153). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-56258-2_13

Johannes Pfau received the M.Sc. degree in Electrical Engineering and Information Technology from Karlsruhe Institute of Technology, Karlsruhe, Germany, in 2017. Since 2017, he is pursuing his Ph.D. at the Institute for Information Processing Technologies at Karlsruhe Institute of Technology. His research focuses on the integration of novel transistor and memory technologies in system design, in particular for reconfigurable systems. His research interests further include architecture changes for power optimization in these systems, as well as the tools and tool flow required to synthesize HDL code for those.



Peter Wagih received the B.Sc. degree in Electronics Engineering and Technology from the German University in Cairo, Egypt, in July 2021. During his undergraduate study, he worked as a junior teaching assistant and a researcher. In 2020, he did his Bachelor's Thesis in manual placement of V-FPGA architecture at Karlsruhe Institute of Technology, Germany. Since August 2021, he works as a Quality Assurance Engineer at Mentor Graphics, a Siemens Business, in Cairo. His interests in research extend to FPGA prototyping of functions concerning machine learning and information security.

Jürgen Becker received the Diploma and Ph.D. (Dr.-Ing.) degree from Technical University Kaiserslautern, Germany. He is full professor for embedded electronic systems and Head of the Institute for Information Processing Technologies (ITIV) at the Karlsruhe Institute of Technology (KIT). From 2005–2009 he has been appointed as Vice President for Education at Universitaet Karlsruhe (TH) and Chief Higher Education Officer (CHEO) at KIT from 2009–2012. Since 2012 till 2014 he served as Secretary General of CLUSTER, an association of 12 leading Technical Universities in Europe. His research interests include Hardware/Software Systems-on-Chip (SoC), Cyber-Physical Systems (CPS), Heterogenous Multicore (MC) Architectures and Design Methods, Reconfigurable Computing, with application in Embedded Systems (Automotive, Industry 4.0, Avionics, HPC Scientific Applications and Experiments). He authored more than 400 papers in peer-reviewed international journals and conferences. Prof. Becker is active in numerous international conferences, as Chairman in TPC & Steering Commitees, e. g. IEEE ISVLSI, IEEE SOCC, RAW, FPL, PATMOS, IFIP VLSI-SoC, DATE, SBCCI, ARC, FCCM, FPT, among others.