Synthesizable FPGA Fabrics Targetable by the Verilog-to-Routing (VTR) CAD Flow

Jin Hee Kim and Jason H. Anderson
Dept. of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
Email: {kimjin14,janders}@ece.utoronto.ca

***Abstract*—We consider implementing FPGAs using a standard cell design methodology, and present a framework for the automated generation of synthesizable FPGA fabrics. The open-source Verilog-to-Routing (VTR) FPGA architecture evaluation framework [1] is extended to generate synthesizable Verilog for its in-memory FPGA architectural device model. The Verilog can be synthesized into standard cells, placed and routed using an ASIC design flow. A second extension to VTR generates a configuration bitstream for the FPGA; that is, the bitstream configures the FPGA to realize a user-provided placed and routed design. The proposed framework and methodology opens the door to silicon implementation of a wide range of VTR-modelled FPGA fabrics. In an experimental study, area and timing-optimized FPGA implementations in 65nm TSMC standard cells are compared with a 65nm Altera commercial FPGA.**

I. Introduction

Standard cell design methodologies are prevalent in the design of modern digital ICs, owing to the high costs associated with manual layout and increasingly complicated design rules in deep sub-100nm technologies. Entire processors [2] and other digital blocks such as PLLs [3] are nowadays mainly synthesized from RTL, as opposed to hand designed at a lower level of abstraction. Field-programmable gate arrays (FPGAs) are one of the few remaining classes of digital IC incorporating a considerable amount of custom layout. The core logic and interconnect tiles in commercial FPGAs are laid out manually, motivated by intense pressure to optimize area, delay and power in the underlying circuitry, as such tiles are stamped out hundreds-to-thousands of times on each die. In this paper, we consider implementing FPGAs in standard cells and assess the gap between a synthesized standard cell and a full custom commercial FPGA implementation.

To realize a standard cell FPGA implementation, we have developed a synthesizable FPGA fabric *generator* within the open-source Verilog-to-Routing (VTR) [1] toolsuite from the University of Toronto. VTR is capable of modelling and mapping circuits into a wide variety of different FPGA architectures. Our generator produces synthesizable Verilog for VTR's in-memory FPGA device model. As such, our generator is not locked into a single FPGA architecture, but rather, is able to produce Verilog for a spectrum of different FPGAs, for example, with different numbers of look-up-tables (LUTs) per logic block, different numbers of tracks per routing channel, or even different switch block connectivities. In addition to producing synthesizable Verilog, we have also extended VTR to produce a configuration bitstream for a user design implemented within the synthesizable FPGA. While the conventional approach used by commercial vendors involves adding CAD support for each new FPGA device; in our case, we have built "silicon support" for an existing and well established FPGA architecture/CAD evaluation toolsuite – VTR.

In addition to the advantages associated with synthesis vs. custom layout, the proposed synthesizable FPGA fabric generator offers a number of benefits. First, it enables VTR-modelled FPGAs to be realized in silicon, democratizing access to FPGA technology. Specifically, our VTR-based approach circumvents a major impediment to the development of new FPGAs, namely, the complexity and cost associated with building CAD tools that can map user circuits into them. Second, the synthesizable FPGAs can be easily ported to new process technologies, by re-synthesizing using a new cell library. Third, the FPGA fabrics we generate are straightforward to incorporate into an SoC; the FPGA module can be instantiated within the surrounding circuitry, and the layout shape/aspect ratio of the FPGA tiles can be tailored according to the overall SoC floorplan.

We synthesize FPGA fabrics into TSMC 65nm standard cells. Through constraints supplied to the ASIC design tools (Synopsys Design Compiler and Cadence Encounter), we produce area-optimized, timing-optimized and balanced FPGA fabric implementations. In an experimental study, we supply VTR with an architecture model closely resembling Altera's Stratix III device, and compare the area and delay of the synthesized standard cell FPGA with Stratix III, which is also implemented in 65nm. The contributions of this paper are:

1) An FPGA fabric generator, built within VTR, capable of producing synthesizable Verilog RTL for a variety of architectures.
2) A configuration bitstream generator for the synthesizable FPGAs.
3) An area/performance comparison between several synthesized standard cell FPGAs and, to the authors' knowledge, the first published study comparing a full-custom commercial FPGA with a synthesized standard cell FPGA.

The remainder of this paper is organized as follows: Section II describes related work and provides background for the subsequent sections. The VTR-based synthesizable fabric and bitstream generation is introduced in Section III. Section IV describes the ASIC flow we used to produce a standard cell implementation. The experimental study appears in Section V. Conclusions and future work are offered in Section VI.

II. Background and Related Work

A. Verilog-to-Routing (VTR)

VTR [1] is an open-source FPGA architecture evaluation/CAD framework from the University of Toronto, comprising of RTL synthesis, logic synthesis, packing, placement, routing and timing/power analysis, as shown in Fig. 1. The inputs to VTR are: 1) a description of an FPGA architecture, and 2) an application benchmark for implementation in the

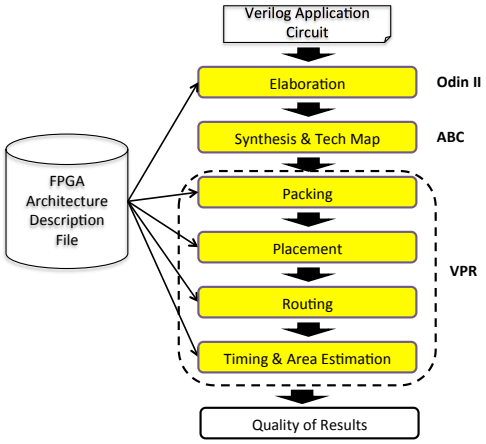

Fig. 1. Verilog-to-Routing flow.

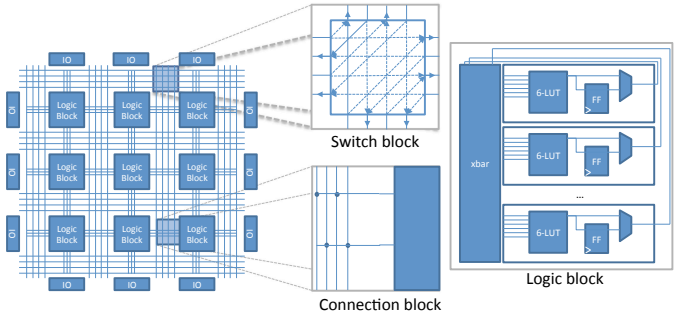

Fig. 2. FPGA architectural components.

FPGA. The architectural description is written in human-readable XML, and through this, an architect can specify both the interconnect and logic architecture of the target FPGA. VTR's internal CAD algorithms are "generic" in the sense that specific architectural details are not hard-coded into the algorithms themselves – the algorithms are designed to do a reasonably good job implementing the application benchmark in a range of architectures. Note that prior to the current work, the VTR flow terminated at the routing stage; it was not possible to realize a silicon implementation of a VTR-modelled architecture. Our work extends VTR to produce synthesizable Verilog for VTR's in-memory architectural device model, as well as a bitstream for the application benchmark implemented in the device.

B. FPGA Architecture

VTR is able to model *island-style* FPGAs [4], a two dimensional array of logic blocks with horizontal and vertical routing channels, surrounded by a ring of I/Os. The key architectural components necessary to understand this paper are shown in Fig. 2. Switch blocks allow horizontal and vertical routing tracks to be programmably connected with one another; connection blocks allow logic block pins to connect to adjacent routing tracks. Logic blocks generally contain one or more look-up-tables (LUTs) and flip-flops (FFs), and an internal crossbar for making local connections.

With respect to routing, VTR allows one to change the number of tracks per channel, wire directionality, the wire segment lengths and relative frequency of wires of a given length, the connectivity between horizontal and vertical wires, and the way wires connect to logic block pins. For logic, VTR permits heterogeneity, where columns of blocks may be of different types; for example, LUT-based soft logic blocks, DSP blocks, and memories. Within each of these types, an architect has a wide range of choices. For example, with soft logic blocks, one can vary the # of LUTs/block, whether the LUTs are fracturable [5] vs. non-fracturable, the richness of the internal local crossbar, the number of FFs, and so on. VTR also supports the notion of *modes*, which represent mutually exclusive ways in which a block may function. For example, a fracturable LUT may operate in single-output mode (implementing a single logic function) or dual-output mode (implementing two logic functions).

C. Related Work

Several recent works bear similarity to our own in that they propose to synthesize FPGA fabrics targetable by VTR. Chaudhuri *et al.* [6] focuses on embedding a reconfigurable FPGA in a system-on-chip (SoC), and enhance the area and performance through floorplanning [7]. Liu [8] studies the impact of the FPGA architectural parameters on the synthesized components of the FPGA. In both of these works, there is little detail on the issues that arise from using ASIC design tools. Moreover, none of these works show a suite of benchmark designs being verified as functional within the synthesized fabric, nor do they compare the synthesized standard cell implementation with a commercial FPGA.

In another work, Aken'Ova [9] investigated island-style FPGAs and improved area and delay gap by using "tactical cells" [10] and floorplanning [11]. The author thoroughly describes architecture changes and solutions to overcome ASIC design flow problems. However, there is little discussion on the generation of the architecture and bitstream.

Other work has focussed on standard cell implementations of application-specific FPGA architectures. An early work by Phillips and Hauck [12] synthesized the reconfigurable-pipelined datapath (RaPiD) [13] architecture using standard cells. The authors observe that customizing the architecture for domain-specific applications, as well as including some FPGA-specific standard cells into the library improves area and performance. Kafafi *et al.* [14] synthesizes a combinational and directional architecture and reports a large area difference relative to a custom-layout design. In work by Wilton *et al.* [15], the authors synthesize a datapath-oriented FPGA fabric with a directional routing architecture. Unlike these past works, which deal with non-standard FPGA architectures, we focus on architectures that resemble today's commercial FPGAs and that are already supported by the VTR framework.

III. Architecture and Bitstream Generation

VPR [16] is the portion of the VTR flow that performs packing, placement and routing. From the user-supplied architectural description, VPR builds an in-memory representation of the entire FPGA device, including all logic and interconnect. The packing, placement and routing steps in VPR implement the application benchmark in the in-memory FPGA device model. Our synthesizable Verilog generator is built within VPR and executes at the end of the routing step. Essentially, our generator code "walks" the in-memory device model to produce synthesizable Verilog, and likewise, by examining the application benchmark's implementation in the device, we produce a configuration bitstream for the FPGA. We elaborate on these steps below.

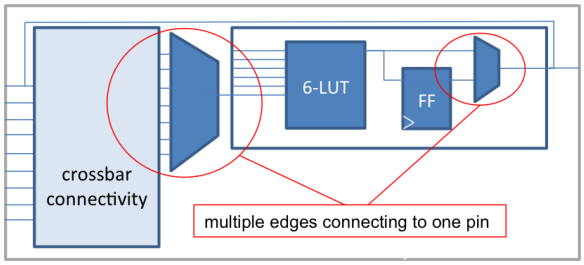

Fig. 3. MUX inference within a logic block.

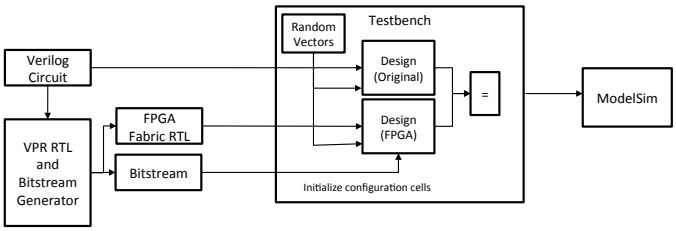

Fig. 4. Synthesizable FPGA verification flow.

A. Generating Synthesizable Verilog

As a first step, we hand-wrote Verilog for two FPGA primitives: a FF, and a Stratix III-like fracturable LUT (see Section V). Subsequently, we *automatically* generate Verilog for the entire FPGA device, a structural netlist of these primitives, as well as other primitives which are generated by our generator code: multiplexers (MUXs) of any size, LUTs with any number of inputs. The generation must handle the following: logic blocks, intra-logic block routing, inter-logic block routing, and configuration cell memory.

Logic Blocks: Logic blocks in VPR are represented in memory as a tree; the tree root represents the entire logic block, nodes at intermediate levels of the tree represent levels of hierarchy in the block, and the leaves represent the primitives (LUTs and FFs). We generate the Verilog for each logic block by first traversing to the leaf nodes. We then move up the tree and, as we visit each node in the hierarchy, its child nodes are defined and instantiated in the output Verilog. The Verilog generated for a logic block has the same hierarchy specified by the architect in the architecture file.

Intra-Logic Block Routing: Routing within a logic block is stored in memory as a graph, where nodes represent pins (on primitives or on intermediate levels of hierarchy) and directed edges represent connections between pins. For a given pin, if there is more than one incoming edge, a routing MUX is inferred. The select inputs to the MUX will be driven by configuration cells (discussed below). Fig. 3 highlights examples of routing MUX inference within a logic block. Crossbars with varying degrees of connectivity can be generated, since VPR only creates edges in its in-memory model for those connections that exist.

Inter-Logic Block Routing: Routing that connects the logic blocks is likewise represented in memory as a graph. In this case, the nodes represent the wire segments and pins. Edges represent programmable connections between such conductors. As above, where there exists more than one edge to a node, MUXs are inferred. These MUXs correspond to the connectivity within switch blocks and connection blocks (Fig. 2). VPR does not model the inter-logic block routing hierarchically – there is no notion of switch block or connection block within VPR's in-memory model. Consequently, each MUX is instantiated in our Verilog without hierarchy.

Configuration Cells: As MUXs that implement programmable connectivity are being instantiated, configuration cells that drive their select inputs must also be instantiated and attached accordingly. We use "fully encoded" MUXs, meaning, a 4-to-1 MUX will have two configuration bits. Other styles of MUX (e.g. flattened MUXs that use more configuration cells and have fewer levels from input-to-output) are left to consider in future work. We use a FF to implement each configuration cell. Then, the cells are connected in a chain, like a shift register. Similarly, for the LUTs in logic blocks, we instantiate configuration cells to hold the LUT's truth table contents.

B. Bitstream Generation

The configuration bitstream is an ordered sequence of `0`'s and `1`'s that configures the FPGA according to the implementation of the application benchmark. Since the configuration cells are connected together in a chain, the `0/1` values shifted in for the benchmark's implementation must align exactly with the ordering of cells in the chain. Thus, to create the configuration bitstream for a design, our generator walks the device model in precisely the same order as is used to generate the synthesizable Verilog. The in-memory implementation of the benchmark is used to assign `0/1` values in the bitstream. For example, consider a 4-to-1 interconnect MUX whose inputs are numbered 0, 1, 2, 3. The path selected through the MUX will be controlled by two configuration cells. Assuming that VPR has routed a signal through input #1, the two configuration cell values in the stream will be `01`. Regarding bitstream generation, there were two challenges worth highlighting discussed below.

Input and Output Equivalence: VPR supports input and output pin equivalence (essentially "pin swapping"). This means that as we generate the bitstream, we have to account for any change in the ordering of the inputs or outputs that may have occurred during routing. For example, consider a MUX within the intra-logic block crossbar. At the packing stage, VPR may have used the i^{th} input to the MUX for a logic signal; however, the VPR router may end up instead using the j^{th} input for the signal (e.g. for timing/routability reasons). During bitstream generation, we account for such changes by examining the routing paths actually used by nets and do not rely on the packed (pre-routed) netlist.

Fracturable LUTs: When LUTs are not fracturable, we may assume that unused inputs are grounded and we configure the LUT truth table accordingly. However, with fracturable LUTs, we must account for inputs that are shared between the LUTs. For example, fracturable LUTs in Altera commercial devices have 8 inputs, where two inputs are shared between the two LUTs. When one of the shared inputs is used in the first LUT, but unused in the second LUT, we can no longer assume that input to be grounded when we specify the truth table for the second LUT. The truth table for the second LUT must be set in such a way that the unused input is a "don't care": the LUT function must be correct regardless of whether the unused input is a `0` or a `1`. This involves replicating the truth table contents for both possible logic states of the unused input.

C. Functional Correctness

Fig. 4 shows the verification flow. We developed a testbench wherein the original application benchmark RTL is simulated in ModelSim with random vectors. Within the same

testbench, the FPGA device RTL, configured with the generated bitstream, is simulated with the *same* random vectors. Output values are checked for equality with each vector applied. Note that this verification flow was used to check correctness at all stages of the standard cell implementation: RTL generated by our VPR generator, post-technology mapping with Synopsys (discussed below), and post-layout with Cadence (also discussed below).

D. Supported Architectures

Presently, our tool is able to generate synthesizable Verilog for FPGAs comprised of LUT/FF-based logic blocks, interconnect and I/Os. Support for other types of blocks, such as DSP or RAM blocks, is left as future work. We support LUTs that are either fracturable or non-fracturable. In fact, LUT fracturability is the only form of VTR "modes" supported by our tool. The modes feature in VTR allows an architect to describe mutually exclusive functionality for a given block. The specification of modes does not contain information regarding how such functionality should be implemented in hardware, nor is it obvious how it could be inferred automatically by a tool such as ours.

Aside from these limitations, our tool supports Verilog/bitstream generation for *all* VTR-targetable architectures – made possible by the approach described above, which walks VTR's in-memory device model. For example, we are able to handle: any # of LUTs/logic block, any switch block/connection block connectivity, wire segments of various lengths, fully or partially populated crossbars within logic blocks.

IV. Standard Cell ASIC Implementation

We use an ASIC design flow to synthesize, place, route, and analyze the circuit, as summarized in Fig. 5. We used Synopsys Design Compiler to synthesize the FPGA to standard cells. Cadence Encounter is used for placement and routing. Synopsys PrimeTime is used for timing analysis.

A. Synthesis to Standard Cells

We evaluated several different synthesis strategies: top-down, "uniquify", or bottom-up. The top-down method is a push-button approach where there entire design is synthesized in "one shot". However, since it processes the whole design at once, it is too run-time and memory intensive to be a viable approach for a large design. In fact, for a 20×20 FPGA with 300 tracks per channel, Design Compiler could not successfully synthesize using the top-down approach. The uniquify approach allows one to break up the design and compile each instance separately. This approach worked, however, it is again run-time intensive, as each instance of the same Verilog module (e.g. a 6-LUT) is compiled individually. We therefore chose the bottom-up approach, in which each required Verilog module is synthesized just once, and the synthesized instances are stitched together to compose the overall synthesized design.

While the bottom-up method produces a more regular implementation and brings run-time benefits, its weakness is that each type of module is synthesized in isolation; i.e. outside of the context of the other modules it connects to when instantiated in the overall FPGA. For example, consider that for a length-16 wire, it may be truncated at the edge of the FPGA, depending on the location from which it is driven. Length-16 wires truncated at different points will all exhibit different load capacitances, and it is undesirable to synthesize a separate/different driver to be used for each variant of

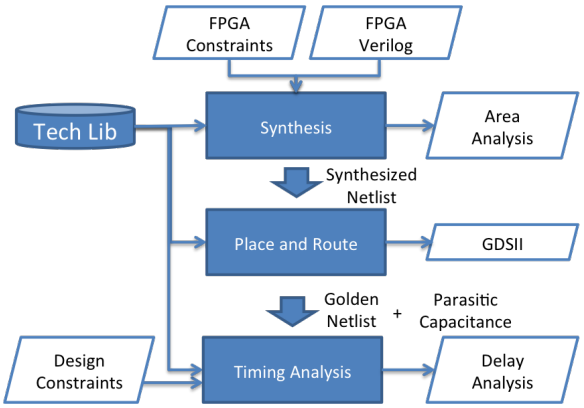

Fig. 5. ASIC design flow.

truncated length-16 wire. Thus, to handle these issues that arise from routing MUXs driving various-length wires, we did the following: 1) we synthesize a single MUX of each size; 2) we insert a fixed-size buffer[1] on the output of each MUX to create a consistent load on the MUX output; and 3) we insert a fixed-size buffer every 2 tiles on inter-logic block interconnect wires, ensuring a roughly uniform load for each buffer.

Design Compiler accepts area and timing constraints, permitting one to trade-off performance vs. area for a single RTL design by changing constraints. In our experimental study (Section V), we have synthesized area-optimized, timing-optimized and balanced FPGAs. Optimizing for area is straightforward: we direct Design Compiler to achieve a target area of 0. Optimizing for timing is more involved, owing to the fact that FPGAs contain many combinational loops before being programmed. Such loops are problematic for timing analysis, and they must be "broken" prior to timing-constrained synthesis. The loops exist within both inter- and intra-logic block routing, and combinations of these. Fig. 6 and Fig. 7 show examples of combinational loops and how we break such loops (via generated constraints provided to Synopsys). In essence, after breaking such loops and by using the bottom-up synthesis approach, we are able to produce a timing-optimized implementation of each module; however, all possible timing paths through the overall FPGA (i.e. across modules) are not optimized globally. Nevertheless, results in the next section demonstrate significantly improved performance in the timing-optimized implementations. Note that timing constraints are only applied to paths through which logic signals may propagate in an application implementation. We do not apply timing constraints to the configuration cells, or paths to/from such cells. The content of such cells only changes when the device is configured; hence, they are not performance critical.

B. Place and Route

Placement and routing proceeds in a flat manner, allowing optimization across the module boundaries. To help the placer, we guide our design using floorplanning. By default, we set floorplanning constraints assuming a chip aspect ratio of 1 (square die) and 85% utilization (as Kuon and Rose discussed [17]). Note that total cell area is known after synthesis to standard cells, making it possible to define a die size with any given utilization ratio.

[1]The Cadence Encounter router also has capabilities for automatic buffer insertion (command `optDesign`), however, because of the size of the design being placed and routed, the router-based buffer insertion repeatedly crashed on our server. We therefore opted to insert buffers during synthesis.

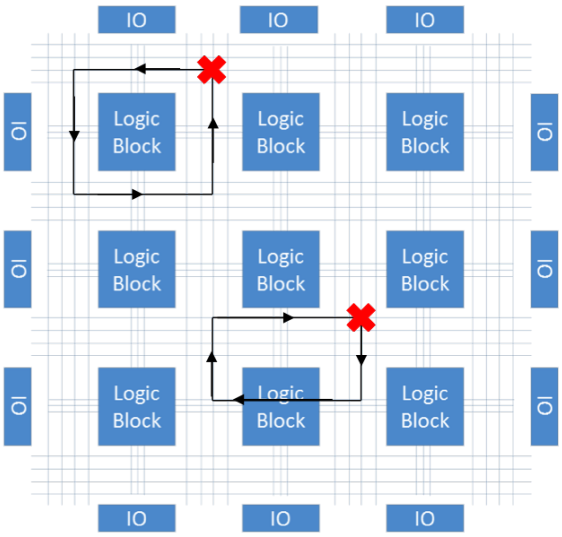

Fig. 6. Combinational loop in inter-logic block routing.

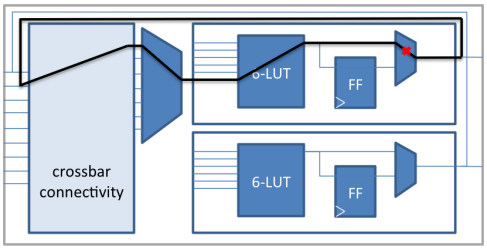

Fig. 7. Combinational loop in logic block.

We found that floorplanning was mandatory to ensure that the physical layout of logic and routing tiles, in terms of ordering in the horizontal and vertical dimensions, matched with that assumed by VPR. Without this, Encounter produced layouts where, for example, logic blocks that VPR saw as adjacent, were actually placed far apart in the layout. Fig. 8 is an example of how configuration cells will drift towards each other due to their connectivity and how two logic blocks that are intended to be adjacent to one another can get separated. For floorplanning the individual modules, we evenly divide up the chip and constrain our logic blocks and the connection MUXs connected to these logic blocks in the appropriate areas. On top of this grid, we overlay another grid to floorplan the switch MUXs in the appropriate areas. The Cadence placer allows one to control the rigidity of the floorplanning constraints, specifically, whether cells are allow to enter/exit each floorplanning region. We set this to the most flexible scheme possible, where the floorplanning constraints are used as a guide to the placer, but cells may exit/enter the specified regions. All of the floorplanning `TCL` commands are *automatically* generated at the same time Verilog description of the FPGA is generated.

Once the designs have been placed and routed, parasitic capacitances are extracted for use by PrimeTime to obtain accurate post-layout timing analysis. Also, at this point, a GDSII file can be written that contains all the mask information.

C. Timing Analysis

Synopsys PrimeTime is used for post-layout timing analysis of: 1) specific paths within the implementation, or 2) an application benchmark programmed on the FPGA. PrimeTime accepts as input the design, annotated with parasitic capacitance information, as well as an SDC (Synopsys design constraints) file. The SDC file specifies which timing paths should be ignored. For 1), we ignore all paths but the specific paths we wish to analyze (see next section) and run timing analysis to obtain their delay. For 2) finding the critical path of an application benchmark implemented within the fabric, the process is more involved. Commercial FPGA vendors provide static timing analysis tools that analyze the performance for user designs implemented in their FPGAs, using delay models of the underlying fabric. To mimic the behavior of such tools for an application implemented within our synthesized fabric, we devised the following approach: during bitstream generation (Section III-B), we have precise knowledge about which FPGA resources are used vs. unused. For each *unused* resource, we automatically generate an SDC constraint to disable timing analysis through the resource. When PrimeTime is invoked to analyze performance of the FPGA device configured with the application bitstream, PrimeTime "sees" only those paths in the *used* part of the FPGA (which should be free of combinational loops, assuming well-designed circuits). The critical path reported by PrimeTime is then analogous to that reported by the timing analysis tools of commercial FPGA vendors. It is important to note that once the FPGA device has been synthesized, placed and routed, timing analysis can be done for any application benchmark by providing PrimeTime with the bitstream and SDC file for that benchmark. Meaning, it is not necessary to synthesize, place and route the FPGA device on an individual benchmark-by-benchmark basis.

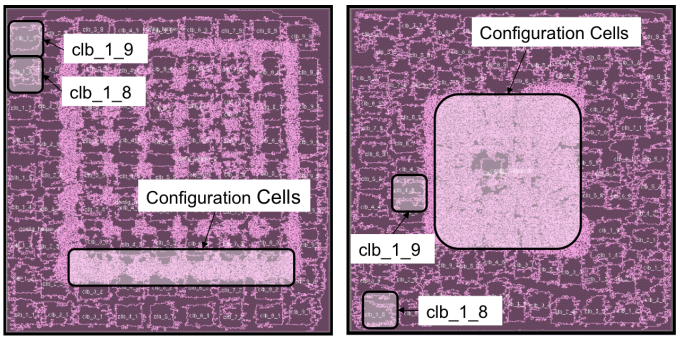

Fig. 8. Floorplanned (left) and unfloorplanned (right) layouts.

A challenge we had to deal with regarding PrimeTime arose due to our bottom-up synthesis strategy and the delay model of the standard cells. PrimeTime reported warnings (RC-009) that in some cases, timing results may be inaccurate as cell drive resistance was too small in comparison with the impedance of the driven network. Recall that in the bottom-up synthesis style, in some cases, Synopsys technology mapping must select cells of a certain size without global context/knowledge of the total RC load driven by such cells. This mainly occurred for large cells driving long interconnect wires, and we were able to eliminate all warnings through the buffer insertion discussed previously.

V. Experimental Study

Table I summarizes the parameters of the FPGA architecture we synthesized into commercial TSMC 65nm standard cells. The architecture is designed to resemble Altera's Stratix III FPGA, which is also fabricated in TSMC's 65nm process, allowing us to make a (roughly) apples-to-apples comparison. The architectural parameters are from a recently

Parameters	Values
FPGA dimensions	20 x 20
K, LUT size	6
N, # of LUTs/logic block	10
Crossbar connectivity	50%
L, Wire length	4 (87%), 16 (13%)
W, Channel width	300
Fc_{in} Input connectivity	0.055
Fc_{out} Output connectivity	0.1

TABLE I. FPGA ARCHITECTURE PARAMETERS.

published Stratix IV architecture capture by Murray *et al.* [18], where authors attempted to model Stratix IV within VTR[2]. Our synthesized FPGA has dimensions of 20 $\times$ 20 logic blocks, with 10 fracturable LUTs/block. There are 300 routing tracks/channel, where 87% of tracks span 4 tiles, and 13% span 16 tiles. Fc_{in}/Fc_{out} refer to the fraction of adjacent tracks a logic block input/output pin may programmably connect to. Within the logic block, the crossbar is 50% populated. We are using fracturable 6-LUTs with 8 inputs, which implies 2-shared inputs in dual-output mode, similar to the extensive architecture described in [19]. Such LUTs can implement any single function of up to 6 variables, or any two functions that together, use no more than 8 unique variables. We reinforce that although in this study we focus on a particular synthesized fabric comparable with Stratix III, our generator is able to automatically produce RTL for a variety of VTR-supported architectures.

We synthesized three variants of the architecture described above: *area-optimized*, *timing-optimized* and *balanced*. For area-optimized, we directed Synopsys to minimize area and imposed no timing constraints. For the timing-optimized, we conversely directed Synopsys to minimize delay, and imposed no area constraints. For the balanced, we took the mid-point of the achieved delays between the area and timing-optimized and set these as the target delays for Synopsys. Fig. 9 shows one of the synthesized FPGA fabric layouts.

In a first set of experiments, we examine the area and performance (of specific paths) of the synthesized FPGA and compare with analogous area and performance data for Stratix III. This first set of experiments is thus *agnostic* to any particular application design being implemented within the fabric – it is a fabric-to-fabric comparison. In a second set of experiments, we compare the performance of application benchmark designs implemented on our fabric to those same designs implemented on Stratix III.

We consider various combinational and sequential benchmarks from the MCNC benchmark suite [20]. Since we are using the full VTR flow, we omitted some designs from the 20 largest MCNC benchmarks where VTR swept away unconnected nodes (as these circuits caused problems for our verification flow which relied on I/O matching). In addition to the MCNC circuits, we added a finite state machine (FSM) that detects a pattern, and also an adder connected to a shift register. These latter two circuits were used mainly for debugging purposes. We use the MCNC circuits in this initial study, as these can be simulated with random vectors and verified with the flow in Fig. 4. Other benchmark suites, such as the VTR suite, contain DSP blocks and RAMs, and are more challenging to simulate/verify, owing to the circuits having reset/control inputs.

[2]While Stratix IV is on a more advanced process than Stratix III, the soft logic block and routing architectures are similar.

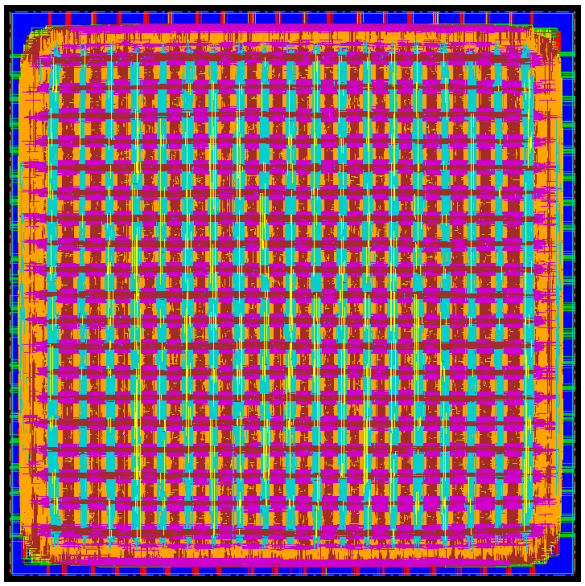

Fig. 9. Synthesized FPGA.

A. Area Analysis

We compare the tile area of our synthesized FPGA to Altera's Stratix III. The tile area of our FPGA was obtained by dividing total die area by the number of logic blocks (20 $\times$ 20 = 400). Table II summarizes the tile area of the three architectures. Stratix III LAB tile area is reported to be $0.0221m\text{m}^2$ by [21]. The area-optimized fabric resulted in the smallest tile area of $0.0316m\text{m}^2$, which is 1.5$\times$ bigger than Stratix III. As expected, the timing-optimized and balanced fabrics were larger: 2.9$\times$ and 1.9$\times$ bigger than Stratix III, respectively. We were encouraged by the area of the synthesized fabrics, especially the area-optimized, which is relatively close to Stratix III.

A number of factors contribute to the area difference vs. Stratix III. First, there are architectural differences. For example, our architecture does not support carry-chains nor are our MUXs fully-decoded. Second, our implementation uses only those standard cells in the TSMC library. In commercial FPGAs, pass-transistors or transmission gates are commonly used to implement MUXs and LUTs; however, we use full CMOS implementations of these primitives. Likewise, we are also using FFs for the configuration cells rather than SRAM cells (as we expect is done in a commercial device). Perhaps most importantly, the Stratix III LAB is custom laid-out.

Delving further into the area results, Fig. 10 shows the breakdown of area into logic, inter- and intra-logic block routing, and configuration for each fabric type. In area-optimized design, configuration cells built of costly FFs in our case, occupy a large portion of the area: 42% of the total. It is likewise not surprising that routing comprises 50% of the fabric area, since we are using standard cell-based MUXs, instead of pass-transistor-style MUXs.

In the timing-optimized FPGA fabric, we observe that configuration cells are reduced to 21% of the total area. This is because the configuration area is kept constant by applying no timing constraints to the configuration cells (they are not performance critical). Routing area has increased to 67% of the area and logic area increased to 13%. Remember that in the timing-optimized fabric, we inserted extra buffers on the inter-logic block wires. However, buffer area is not appreciable: 2% of the total.

FPGA Fabric	# of Std. Cells	Total Area (mm^2)	Tile Area (mm^2)
Area-Optimized	3,577,520	12.65	0.0316
Timing-Optimized	7,521,616	25.72	0.0643
Balanced	5,298,588	16.89	0.0422

TABLE II. AREA OF SYNTHESIZED FPGA.

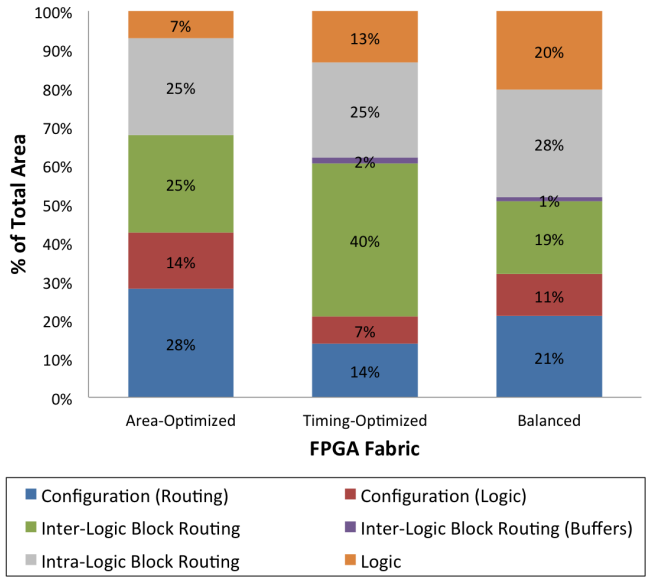

Fig. 10. Area breakdown.

In the balanced FPGA fabric, both timing and area constraints were applied, however, we give a more relaxed timing constraint to the routing circuitry to save area. This leads to logic taking up 31% (logic + config for logic) and routing taking up 69% (routing + config for routing + buffers) of the total area. Note that in the balanced fabric, we keep the LUTs timing constraint aggressive, since the LUT takes up a small portion of the total area.

B. Timing Analysis

We first examine the delay of commonly-used paths in the synthesized fabrics and Stratix III (application-agnostic analysis). Specifically, we looked at the following three paths:

1) L_0: FF $\rightarrow$ crossbar $\rightarrow$ LUT $\rightarrow$ FF (within a logic block).
2) L_4: FF $\rightarrow$ length-4 wire $\rightarrow$ crossbar $\rightarrow$ LUT $\rightarrow$ FF (a path of length 4).
3) L_{16}: FF $\rightarrow$ length-16 wire $\rightarrow$ crossbar $\rightarrow$ LUT $\rightarrow$ FF (a path of length 16).

Table III is a summary of average delay of these paths in 6 different areas of the FPGAs (in the four corners of the fabric, and also on the middle of the left/right sides). In the synthesized fabrics, we manually selected the 6 paths by creating an SDC file that reports the delay for each. In doing so, we are assured that our analysis reflects the use of a length-4 or length-16 wire, accordingly. For the Altera Stratix III delays, we use Altera's LogicLock feature to place two connected flip-flops 4 or 16 logic blocks away from one another, and then use Altera's TimeQuest tool ascertain the path delay. For our fabrics, within the logic blocks, we apply an SDC constraint so that we measure the path corresponding to the fastest LUT input; this is to be comparable to Altera,

FPGA Fabric	L_0 (ns)	L_4 (ns)	L_{16} (ns)
Area-Optimized	3.71	7.38	17.31
Timing-Optimized	1.79	2.90	4.92
Balanced	1.34	3.73	7.32
Stratix III	0.73	1.03	1.54

TABLE III. ARCHITECTURE DELAY.

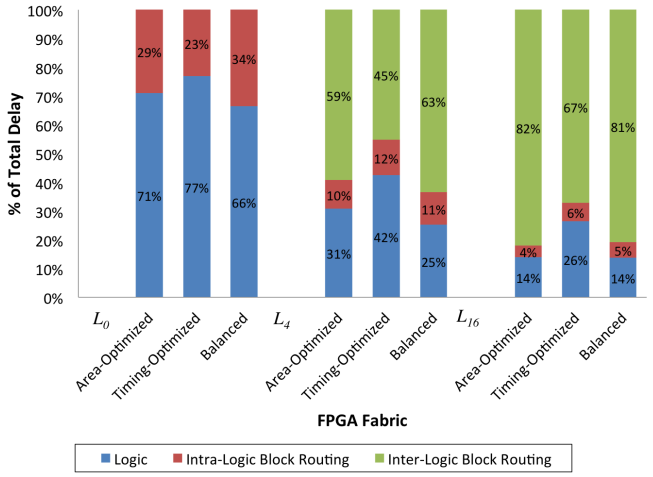

Fig. 11. Delay breakdown.

since Quartus uses the input that results in the smallest delay. Note that for the timing results in this paper, we use the *slowest* speedgrade for Stratix III, and compare with slowest-process-corner analysis for our fabric.

Comparing our timing-optimized implementation to Stratix III, the delay is 2.3× to 3.5× slower, with the gap being larger for longer wire lengths. The balanced fabric is 1.8× to 4.7× slower, and the area-optimized fabric is 5× to 11.2× slower than Stratix III. We believe the reason that the delay gap vs. Stratix III grows with wire length is related to the difficulty in handling long wires in the ASIC toolflow. The relative results between the synthesized fabrics are as expected: the area-optimized fabric is overall slower than the balanced design, and the balanced design is slower than the timing-optimized fabric, except for the L_0 path. L_0 reflects timing within a logic block; the inter-logic block routing MUXs are not included in the delay. The slightly lower delay in the balanced design may be due to the heuristic nature of ASIC mapping, placement, and routing tools. It may also be because in our fabric (unlike Stratix III), one MUX drives the output of a logic block to both feedback and inter-logic block routing paths. That output MUX is timing optimized in our balanced fabric implementation, yet it sees a smaller load than in the timing-optimized fabric implementation. Similar to the area analysis, Fig. 11 shows a delay breakdown for the three types of paths for all architectures. The L_4 and L_{16} fabric delays are dominated by routing delay. This confirms that we need to use optimizations such as buffering to reduce the routing delay.

In the second part of the performance study, we looked at how benchmark circuits perform on the synthesized FPGA vs. Stratix III. The same Verilog file is passed to each tool for implementing the circuits on the FPGAs. We use the SDC file generated with the bitstream to "program" our synthesized FPGA as discussed in Section IV-C. Note that the results of this experiment are not solely reflective of the fabric speed, but also of the differences in architectures, and in the CAD tools supporting the architectures: open-source VTR vs. Altera's

Benchmark Circuits	Area-Optimized	Timing-Optimized	Balanced	Stratix III
alu4	67.80	22.29	32.47	5.293
apex4	74.62	23.06	36.08	5.271
des	68.65	21.58	31.26	6.696
ex1010	103.59	31.36	48.09	7.248
ex5p	68.68	21.86	31.54	5.455
misex3	74.78	22.59	34.32	5.281
pdc	111.02	33.72	51.66	7.213
seq	69.79	22.66	32.66	5.742
spla	112.73	34.92	51.30	6.654
diffeq	69.42	23.28	29.74	4.391
dsip	35.94	11.71	17.75	5.918
elliptic	103.42	32.45	44.02	6.909
frisc	118.72	38.06	52.37	7.865
tseng	60.43	20.05	25.92	4.519
addshift16	37.01	13.43	16.61	4.31
fsm	5.75	1.71	2.85	1.113
Geo. Mean	63.13	20.10	28.97	5.25

TABLE IV. CRITICAL PATH DELAY (nS) OF DESIGNS ON FPGA.

Quartus II. Table IV lists the critical path delays reported by the tools. The reported critical path delays do not include clock skew nor I/O cell delays (only core logic and routing).

In combinational designs (top part of table), both designs were given input-to-output delay constraints. On average, there is a $\sim$3.8$\times$ increase in delay between the timing-optimized and Stratix III FPGAs (see geo. mean row at bottom of table). The delay gap between the two FPGAs increased from our architecture delay study above, likely due to the weaknesses of the open-source VTR flow vs. Quartus II. In the sequential designs (bottom part of table), the critical path delays reported include register-to-register and I/O paths. Most circuits show similar increase in delay as the combinational designs; however, the *dsip* and *fsm* benchmarks showed smaller increases of 1.5$\times$ to 2$\times$. The critical paths of these circuits have fewer logic levels compared to the other designs.

It is worthwhile to mention that one of the key advantages of a *synthesizable* FPGA fabric that is it permits the type of exploration done here: the ability to realize fabrics with different area/delay trade-offs from a single RTL source, simply by changing constraints provided to the ASIC tools. Such an exploration is highly costly if manual layout is required for each fabric.

VI. CONCLUSIONS AND FUTURE WORK

In this work, we propose to *automatically* generate synthesizable FPGA fabrics within the open-source FPGA CAD tool, VTR. The fabrics we generate are then synthesized, placed and routed using a standard ASIC design flow into a commercial standard cell library. We synthesized 3 variants of an FPGA fabric (modelled on Altera's Stratix III) into 65nm TSMC standard cells: timing-optimized, area-optimized, and balanced. We compared the tile area of our smallest FPGA fabric (area-optimized) with Altera's Stratix III and found our fabric used 1.5$\times$ more area. Our timing-optimized fabric required 3$\times$ more area than Stratix III. With respect to performance, the critical paths of designs implemented in our timing-optimized fabric are $\sim$3.8$\times$ longer, on average, than in Stratix III; however, in some benchmarks the delay gap was as low as 1.5$\times$. Overall, we are encouraged by the silicon area and performance of our fabric relative to Altera's, especially considering Stratix III is custom laid-out and undoubtedly highly optimized. To our knowledge, this work represents the first comparison of a standard cell FPGA implementation to a commercial FPGA. The proposed VTR-based synthesizable FPGA generator opens the door to actual silicon implementation of FPGAs targetable by an established CAD tool.

In the future, we would like to assess power consumption, and extend architecture and bitstream generation to accept all architectures supported by VTR, including those with DSP blocks and memories. Further work is also needed to support designs with multiple clocks. Finally, we would like to explore the utility of adding custom library cells that are specifically tailored for FPGAs, particularly for efficient MUX and configuration cell implementations.

REFERENCES

[1] J. Rose *et al.*, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," in *FPGA*. ACM, pp. 77–86.

[2] E. Fluhr *et al.*, "Power8: A 12-core server-class processor in 22nm SOI with 7.6tb/s off-chip bandwidth," in *ISSCC*. IEEE, 2014, pp. 96–97.

[3] W. Deng *et al.*, "A Fully Synthesizable All-Digital PLL With Interpolative Phase Coupled Oscillator, Current-Output DAC, and Fine-Resolution Digital Varactor Using Gated Edge Injection Technique," *JSSC*, vol. 50, no. 1, pp. 68–80, Jan 2015.

[4] V. Betz *et al.*, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[5] "Stratix III ALM Logic Structure's 8-Input Fracturable LUT," Altera Corp., Tech. Rep., 2015. [Online]. Available: https://www.altera.com/products/fpga/features/st3-logic-structure.html

[6] S. Chaudhuri *et al.*, "An 8x8 run-time reconfigurable FPGA embedded in a SoC," in *DAC*. ACM/IEEE, 2008, pp. 120–125.

[7] ——, "Efficient modeling and floorplanning of embedded-FPGA fabric," in *FPL*. IEEE, 2007, pp. 665–669.

[8] H. J. Liu, "Archipelago – An Open Source FPGA with Toolflow Support," Master's thesis, University of California at Berkeley, 2014.

[9] V. Aken'Ova, "Bridging the gap between soft and hard eFPGA design," Master's thesis, University of British Columbia, 2005.

[10] V. Aken'Ova *et al.*, "An improved "soft" eFPGA design and implementation strategy," in *IEEE CICC*, 2005, pp. 179–182.

[11] V. Aken'Ova and R. Saleh, "A "soft++" eFPGA physical design approach with case studies in 180nm and 90nm," in *ISVLSI*. IEEE, 2006.

[12] S. Phillips and S. Hauck, "Automatic layout of domain-specific reconfigurable subsystems for system-on-a-chip," in *FPGA*. ACM, 2002, pp. 165–173.

[13] C. Ebeling *et al.*, "RaPiD Reconfigurable pipelined datapath," in *Field-programmable logic smart applications, new paradigms and compilers*, 1996, pp. 126–135.

[14] N. Kafafi *et al.*, "Architectures and algorithms for synthesizable embedded programmable logic cores," in *FPGA*. ACM, 2003, pp. 3–11.

[15] S. Wilton *et al.*, "A synthesizable datapath-oriented embedded FPGA fabric," in *FPGA*. ACM, 2007, pp. 33–41.

[16] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL*, 1997, pp. 213–222.

[17] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *Trans. on CAD*, vol. 26, no. 2, pp. 203–215, 2007.

[18] K. E. Murray *et al.*, "Titan: Enabling large and complex benchmarks in academic CAD," in *FPL*. IEEE, 2013.

[19] J. Luu, "Architecture-Aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays," Ph.D. dissertation, University of Toronto, 2014.

[20] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. Microelectronics Center of North Carolina, 1991.

[21] H. Wong *et al.*, "Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture," in *FPGA*. ACM, 2011, pp. 5–14.