VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling

KEVIN E. MURRAY, University of Toronto, Canada
OLEG PETELIN*, University of Toronto, Canada
SHENG ZHONG†, University of Toronto, Canada
JIA MIN WANG, University of Toronto, Canada
MOHAMED ELDAFRAWY, University of Toronto, Canada
JEAN-PHILIPPE LEGAULT, University of New Brunswick, Canada
EUGENE SHA‡, University of Toronto, Canada
AARON G. GRAHAM, University of New Brunswick, Canada
JEAN WU, University of Toronto, Canada
MATTHEW J. P. WALKER, University of Toronto, Canada
HANQING ZENG§, University of Toronto, Canada
PANAGIOTIS PATROS, University of New Brunswick, Canada
JASON LUU, Intel Programmable Solutions Group, Canada
KENNETH B. KENT, University of New Brunswick, Canada
VAUGHN BETZ, University of Toronto, Canada

Developing Field Programmable Gate Array (FPGA) architectures is challenging due to the competing requirements of various application domains, and changing manufacturing process technology. This is compounded by the difficulty of fairly evaluating FPGA architectural choices, which requires sophisticated high-quality Computer Aided Design (CAD) tools to target each potential architecture. This article describes version 8.0 of the open source Verilog To Routing

*Currently with Intel Programmable Solutions Group, Toronto, Canada
†Currently with the University of Michigan
‡Currently with Intel Programmable Solutions Group, Toronto, Canada
§Currently with University of Southern Californa, Los Angeles, USA zengh@usc.edu

Authors' addresses: Kevin E. Murray, University of Toronto, Toronto, Ontario, Canada, kmurray@ece.utoronto.ca; Oleg Petelin, University of Toronto, Toronto, Ontario, Canada; Sheng Zhong, University of Toronto, Toronto, Ontario, Canada; Jia Min Wang, University of Toronto, Toronto, Ontario, Canada; Mohamed Eldafrawy, University of Toronto, Toronto, Ontario, Canada; Jean-Philippe Legault, University of New Brunswick, Fredericton, New Brunswick, Canada; Eugene Sha, University of Toronto, Toronto, Ontario, Canada; Aaron G. Graham, University of New Brunswick, Fredericton, New Brunswick, Canada; Jean Wu, University of Toronto, Toronto, Ontario, Canada; Matthew J. P. Walker, University of Toronto, Toronto, Ontario, Canada; Hanqing Zeng, University of Toronto, Toronto, Ontario, Canada; Panagiotis Patros, University of New Brunswick, Fredericton, New Brunswick, Canada; Jason Luu, Intel Programmable Solutions Group, Toronto, Ontario, Canada; Kenneth B. Kent, University of New Brunswick, Fredericton, New Brunswick, Canada; Vaughn Betz, University of Toronto, Ontario, Canada.

1936-7406/2020/0-ART0 $15.00
https://doi.org/

(VTR) project, which provides such a design flow. VTR 8 expands the scope of FPGA architectures which can be modelled, allowing VTR to target and model many details of both commercial and proposed FPGA architectures. The VTR design flow also serves as a baseline for evaluating new CAD algorithms. It is therefore important, for both CAD algorithm comparisons and the validity of architectural conclusions, that VTR produce high-quality circuit implementations. VTR 8 significantly improves optimization quality (reductions of 15% minimum routable channel width, 41% wirelength, and 12% critical path delay), run-time (5.3× faster) and memory footprint (3.3× lower). Finally, we demonstrate VTR is run-time and memory footprint efficient, while producing circuit implementations of reasonable quality compared to highly-tuned architecture-specific industrial tools – showing that architecture generality, good implementation quality and run-time efficiency are not mutually exclusive goals.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Physical design (EDA)**; **Software tools for EDA**; *Logic synthesis*;

Additional Key Words and Phrases: Computer Aided Design (CAD), Electronic Design Automation (EDA), Field Programmable Gate Array (FPGA), Packing, Placement, Routing, Verilog To Routing (VTR), Versatile Place and Route (VPR)

ACM Reference Format:
Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling. *ACM Trans. Reconfig. Technol. Syst.* 0, 0, Article 0 (2020), 60 pages. https://doi.org/

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have compelling advantages as a computational platform, offering the potential for high performance and power efficiency while remaining flexible and re-programmable [94]. This has led to FPGAs being used in a wide range of application domains including telecommunications, high performance computing and machine learning [30, 93].

The application domains targeting FPGAs and the underlying manufacturing process technology used to build FPGAs both evolve rapidly. As a result FPGA architectures can not remain static – they must evolve to optimize their efficiency for the target process technology, and both legacy and emerging application domains. Furthermore novel architectural ideas are constantly being proposed. Being able to quantitatively compare different architectural ideas is key to developing a high quality FPGA architecture.

However performing these comparisons is challenging, as it requires well optimized implementations for a suite of representative benchmark circuits across many architectural variations, along with accurate area, delay and power estimates. This requires a full high quality automated design flow to target each architecture. Since creating such optimization algorithms for a specific FPGA architecture is itself a complex research problem [22], it is impractical to develop unique optimization algorithms for every architectural variant. Similarly, accurately estimating the delay, area and power of a completed design implementation on an FPGA also requires sophisticated algorithms and tools [28, 44, 63, 81].

The approach pioneered by Versatile Place and Route (VPR) [15, 77] and the Verilog To Routing (VTR) project [76, 95][1] is to build highly flexible Computer Aided Design (CAD) tools and optimization algorithms which can target and adapt to a wide variety of FPGA

[1]In this work we use VPR to refer to the architecture modelling and place & route tool, and VTR to refer to the design flow including ODIN, ABC and VPR (Section 3).

architectures. This allows FPGA architects to efficiently evaluate different architectural choices while ensuring well optimized implementations are produced for each benchmark circuit. This facilitates fair comparisons across architectures.

In this work we introduce version 8.0 of the open-source Verilog To Routing project.[2] The main contributions of this work include:

- Extending VTR's FPGA architecture modelling capabilities to capture a broader range of FPGA architectures,
- An improved capture of the Intel Stratix IV FPGA architecture,
- Enhancing VTR's interoperability with other tools and design flows,
- More complete and flexible timing analysis,
- Significant improvements to optimization quality (41% wirelength, 12% critical path delay, and 15% minimum routeable channel width reductions on the VTR benchmarks), robustness (completes 9 more large Titan benchmarks), run-time (5.3× faster on the VTR benchmarks, reducing average run-time on the large VTR designs from over 35 minutes to less than 7 minutes, and from 3.4 hours to less than 40 minutes on the larger Titan benchmarks) and memory usage (3.3× lower on the VTR benchmarks), and
- A detailed evaluation comparing VPR to Intel's commercial Quartus CAD flow which shows, despite its flexibility, VPR 8 is competitive in run-time and reasonable in optimization quality compared to a highly-optimized architecture-specific tool.[3]

In combination, these enhancements mean a wider range of architectures can be explored with both more detailed analysis and higher quality optimization, which increase confidence in the resulting architectural conclusions. VTR 8's more flexible architectural representation also means it can model and target commercial FPGA devices, and program them through architecture specific bitstream generators which translate VTR's implementation results into bitstreams [6, 73].

This paper is organized into two main parts. The first part focuses on the overall VTR CAD flow and its architecture modelling features:

- Section 2 presents related work,
- Section 3 introduces the VTR CAD flow and its extensions,
- Section 4 describes enhancements to VTR's architecture modelling capabilities, and
- Section 5 illustrates how these features enable an improved capture of Stratix IV FPGAs.

The second part focuses on algorithmic and engineering improvements to the CAD flow:

- Section 6 discusses improvements to logic optimization and technology mapping,
- Section 7 studies improvements to packing,
- Section 8 explores improvements to placement,
- Section 9 investigates improvements to routing,
- Section 10 presents improvements to timing analysis,
- Section 11 addresses software engineering concerns, and
- Section 12 performs a detailed evaluation of the VTR 8 CAD flow.

Finally, Sections 13 and 14 present our conclusions and outline future work.

[2]VTR 8 can be downloaded from verilogtorouting.org, corresponding to source code revision `v8.0.0` (`fd69801f1`).

[3]Note that since VTR 7 [76], the VTR and VPR version numbers have been aligned. Therefore VPR 8 refers to the version of VPR used with VTR 8, and VPR 7 to the version used in VTR 7.

2 RELATED WORK

Related work falls into two broad categories: FPGA CAD and architecture exploration frameworks, and research which builds upon and compares to VTR.

2.1 FPGA CAD & Architecture Exploration Frameworks

There have been a variety of FPGA related-frameworks produced which target different aspects of the design implementation and architecture exploration space.

Commercial FPGA vendors have internal tools which they use to develop and architect their next generation devices. For instance, Intel/Altera use their FPGA Modelling Toolkit, originally based on a heavily modified version of VPR, to explore new FPGA architectures [70]. However these tools are closed-source and proprietary – making them unavailable to other researchers.

Torc [104] and RapidSmith [46, 68] provide infrastructure for manipulating the design implementation of commercial Xilinx FPGAs, but both rely on the XDL interface provided by Xilinx's legacy ISE tools. More recently, RapidWright has enabled low-level access to the design implementation of more recent Xilinx FPGAs [67]. However these tools can only target fixed devices from a single manufacturer, making it impractical to perform architecture research within these frameworks. Furthermore, while these tools provide the infrastructure to build CAD tools targeting these devices, they provide simple proof-of-concept algorithms for optimization stages like placement and routing (e.g. non-timing driven, conflict unaware) – making them unsuitable for architecture evaluation and as baselines for comparing CAD algorithms.

Independence [99] is a very general placement and routing tool for evaluating FPGA architectures, but it runs more than 4 orders of magnitude slower than VPR – too slow to allow broad exploration of architectures with large-scale benchmarks.

2.2 Research Using VTR

VTR is commonly used as a baseline upon which other tools and research build. These works broadly fall into three categories: FPGA Architecture, CAD algorithms and hybrid design flows.

VTR's ability to target a wide range of novel FPGA architectures, combined with its adaptive optimization algorithms make it a common choice for exploring and evaluating FPGA architectural ideas. Some examples include:

- Introducing new hard blocks to accelerate dynamic memory access [88],
- Evaluating the trade-offs between Look-up Table (LUT) size and logic block size [130], and LUT fracturablitity and logic block routing connectivity [125].
- Exploring new logic elements such as AND-Inverter Cones [89, 131], NAND-NOR Cones [52], dedicated Muxes [29], and reduced flexibility LUTs [10, 36]
- Optimizing logic blocks for arithmetic operations [78, 83],
- Developing routing architectures which exploit mixtures of routing wire lengths and complex switch block patterns [92],
- Quantifying the impact of 3D stacking [102] and interposer [87] technologies,
- Investigating the efficiency of different RAM block architectures [61], and implementation technologies [126],
- Exploring the impact of different semiconductor technologies including DRAM [42], threshold logic [65], and non-volatile technologies like MTJ, [51], PCM [117] and RRAM [110],

- Evaluating power reduction methods like charge recycling [53], power gating [97] and dynamic voltage scaling [11], and
- Developing uncloneable & secure FPGAs [37].

VTR's CAD algorithms are robust and produce high quality results while still being performant. This makes VTR a common infrastructure for exploring and evaluating new CAD algorithms and optimization techniques. These include new or enhanced algorithms which aim to improve run-time/quality trade offs during packing [27, 114], placement [26, 27, 82, 113, 129], and also include cross-stage techniques like multi-clock timing optimization [116] and CAD parameter auto-tuning [123]. Other approaches include reducing run-time through parallel processing during packing [114], placement [14, 39, 43], routing [49, 50, 100, 105] and timing analysis [81]. VTR has also been used to explore novel CAD techniques like device-aging aware routing [62] and the insertion of debug logic into unused resources [56]. Machine Learning enhanced CAD techniques have also been explored within VTR such as, the estimation of routing congestion early in the design flow [127], and the creation more efficient and adaptive optimization algorithms with Reinforcement Learning [82].

In addition to architecture and CAD research, VTR has also been used to create various hybrid flows which often involve targeting or modelling fabricated FPGA devices and often mix commercial and open-source tools. Some examples include:

- Interfacing with Intel/Altera tools for front-end HDL synthesis [84, 85],
- Programming Xilinx FPGAs by interfacing with Xilinx's tools [54, 55],
- Automatically generating standard cell realizations of VTR generated FPGA architectures [21, 45, 64, 73, 111], and
- Programming fabricated FPGAs using device-specific bitstream generators (which translate VTR's circuit implementation into bitstreams) targeting: realizations of VTR generated FPGA architectures [21, 33, 37, 45, 64, 66, 74, 111, 128], non-VTR generated FPGA architectures [73, 112], and commercial Xilinx & Lattice FPGAs [6].

The VTR code base has also been used by commercial companies such as Intel/Altera and Texas Instruments, as well as numerous FPGA start-up companies – often forming the basis of their internal place and route tools.

Given the many uses of VTR, improving the flexibility of the tool flow, the quality of its results and the software engineering of its code base is very important. Previous work [54, 85] has shown significant gaps in quality and run-time between commercial and open-source/academic FPGA design flows. In addition FPGA sizes continue to grow, putting increasing pressure on CAD flow run-time to allow efficient design of large-scale systems.

This has led some to question whether architecture-flexible tools (like VTR) can be adapted to target commercial devices while generating high quality implementations in reasonable run-time – or whether specialized device-specific tools (either commercial [31, 32] or open-source [1, 98]) are required. In this paper we aim to show that architecture generality, good implementation quality and efficient run-times are not mutually exclusive goals.

3 DESIGN FLOW

The VTR design flow and several of its variants are shown in Figure 1. It provides a full design implementation flow which maps an application onto a target FPGA architecture. Importantly, and unlike commercial FPGA CAD flows, the VTR design flow is flexible and can target a wide range of FPGA architectures. This makes it a useful design flow for evaluating and comparing different FPGA architectures.

The VTR flow takes two primary inputs:

- a human-readable FPGA architecture description, and

- an application benchmark described in behavioural Hardware Description Language (HDL).

The design flow will build a model of the specified FPGA architecture and map the given application onto it.

The design flow proceeds in several stages. First, the behavioural HDL is elaborated and synthesized into a circuit consisting of soft logic and FPGA architectural primitives (e.g. FFs, multipliers, adders) using Odin II [60]. Second, the circuit logic is passed to ABC [18, 109] which performs technology independent combinational and sequential logic optimizations, and then technology maps the soft logic to LUTs. Third, the resulting technology-mapped circuit netlist is then passed to VPR which generates the physical implementation of the circuit on the target FPGA architecture.[4]

VPR first builds a detailed model of the target FPGA device based on the FPGA architecture description. VPR then [17]:

- Packs the circuit netlist into the blocks available on the FPGA device (e.g. Logic/DSP/RAM Blocks).
- Places the clustered blocks onto valid grid locations.
- Routes all connections in the netlist through the FPGA's interconnect network.

At each stage VPR optimizes the implementation for area and speed. Finally, VPR analyzes the resulting circuit implementation to produce area, speed and power results, and a post-implementation netlist.

The tools which make up the VTR design flow can be re-used and re-combined. An increasingly common use case is to use portions of the VTR CAD flow. These use cases often relate to targeting fabricated FPGA devices or mixing open source and proprietary tools. As a result, improving the ability to mix-and-match tools and increase the flexibility of the design flow is an area we have focused on enhancing in VTR 8.

For example, the Titan Flow [84] uses Intel's Quartus [31] FPGA CAD tool to perform logic synthesis, optimization and technology mapping. Through the use of a conversion tool (VQM to BLIF) it is then possible to use the resulting technology mapped netlist with VPR. Other synthesis tools like Yosys [119] can also be used, provided they generate technology mapped netlists with primitives matching the targeted FPGA architecture.

3.1 Design Flow Enhancements

VTR 8 extends the capabilities of the design flow compared to previous releases. These enhancements make it more flexible, and improve interoperability with other tools.

3.1.1 Complete External Implementation State & Flexible Analysis. In addition to support for loading a design's packing or placement from external files, VPR 8 adds support for also loading routing from an external file. This enables the complete design implementation (labeled in Figure 1) to be loaded into VPR from external files. This allows individual optimization stages to be run independently, and allows other tools to modify the design implementation.

This facilitates a variety of alternative design flows, as shown in Figure 1. For instance, tools other than VPR can perform some or all of the design implementation, while still making use of VPR's device modelling or area, timing and power analysis features. Another use case is to re-analyze a fixed design implementation under different operating conditions such as different supply voltages [11].

[4]For details on the file formats used by VTR see the documentation at docs.verilogtorouting.org.

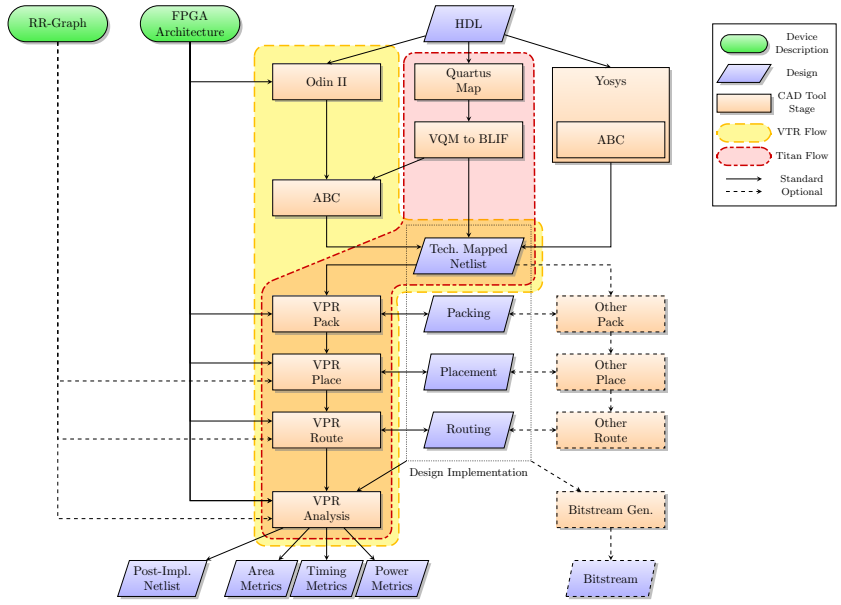
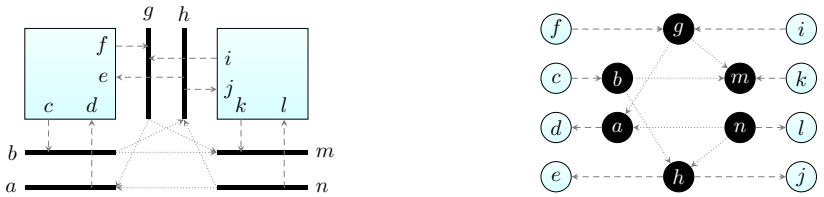


Fig. 1. VTR-based CAD Flow Variants



(a) Routing Architecture (b) Corresponding Routing Resource Graph

Fig. 2. Routing Resource Graph. Dotted arrows represent switch-block switches, while dashed arrows represent connection-block switches.

3.1.2 External Routing Resource Graph. The detailed routing architecture is a key component of any FPGA. VPR models the detailed routing architecture using a Routing Resource Graph (RR-Graph), which represents conductors (wires and pins) as nodes, and the switches between them as edges. The RR-Graph is a key abstraction which allows VPR's optimization and analysis engines to easily target a wide variety of FPGA routing architectures. Figure 2 illustrates how a particular detailed routing architecture can be modelled by an RR-Graph.

Historically, VPR has generated RR-graphs internally from a high-level specification provided in an FPGA architecture description file [16]. Using a parameterizable RR-Graph generator has been very productive for FPGA research as it allows rapid exploration and evaluation of a wide range of routing architectures. While VTR 8 significantly extends the capabilities of VPR's RR-Graph generator (Section 4.2), this approach has limitations when targeting pre-fabricated FPGA devices.

To address these limitations VPR 8 adds support for loading the targeted device's RR-Graph from an external file.[5] This yields several advantages:

[5]The RR-Graph is a low-level detailed description. This makes it highly flexible, but generally not practical to create by hand. Typically it would be created by another program or a layout extractor. To aid in this process, VPR can generate an RR-Graph file from its internal RR-graph representation.

- First, it enables an RR-Graph to be loaded which exactly matches a fabricated device. This is crucial, as any disagreement between the CAD tool's internal device model and the physical device may result in invalid or functionally incorrect bitstreams.
- Second, it allows VPR to target routing architectures with features which are either difficult to specify, or not supported by VPR's RR-Graph generator. Provided an RR-Graph describing such an architecture can be generated (e.g. by another tool) it can be targeted by VPR.
- Third, it decouples the RR-Graph targeted by VPR from VPR's RR-Graph generation algorithm. The routing architecture specification provided to the RR-Graph generator is under-specified, and hence does not uniquely describe a particular architecture. This is highly beneficial since it is easy for the architect to specify, but means the RR-Graph generator has significant freedom to choose how it constructs the routing architecture. As a consequence, the generated routing architecture is dependent on the exact RR-Graph generation algorithm used, and changes to this algorithm (e.g. to add support for new features, or improve the quality of the generated architecture) may change the resulting routing architecture.
- Fourth, the RR-Graph becomes an interchange format describing the FPGA's routing architecture. When committing a VPR generated routing architecture to silicon, the RR-Graph becomes the specification the physical layout must implement. Alternately, an RR-Graph produced from a physical layout becomes the device specification CAD tools (e.g. VPR or other tools like bitstream generators) target.

Support for external RR-Graphs improves VPR's interoperability with other tools, and helps to decouple VPR's device model generation, optimization and analysis stages. For instance, the Symbiflow [6] and PRGA [73] projects generate RR-Graphs in order to target their architectures with VPR.

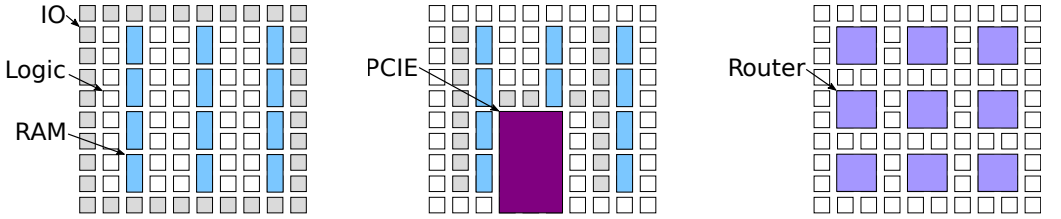
3.1.3 Post-Implementation Netlist & Verification. VPR supports generating a post-implementation netlist which structurally models the design after complete implementation in the FPGA. It can also produce delay information for each primitive and interconnect element in the FPGA. These can be used to simulate the design implementation, with realistic delays, to check its functional correctness.

In VPR 8 this ability has been extended to enable the post-implementation netlist to be produced in BLIF format (in addition to Verilog). This allows ABC to formally verify (prove) the original netlist and final design implementation are logically equivalent. This provides a strong check ensuring the tool flow correctly implements the original design.

4 ARCHITECTURE MODELLING ENHANCEMENTS

FPGA architectures continue to evolve as new application domains emerge, the underlying process technology changes, and new architecture features are proposed. To facilitate exploration of these changing characteristics we have extended and improved VTR's modelling capabilities.

In particular VTR now supports: more general device grids which are not restricted to columns with perimeter I/O (Section 4.1), more general and flexible routing architecture descriptions (Section 4.2), as well as a variety of area and delay modelling improvements (Section 4.3). To illustrate the utility of these enhancements we present a more accurate model of a commercial Intel Stratix IV FPGA architecture in Section 5.



(a) Traditional column-based layout with perimeter I/O (b) Non-column based layout with non-perimeter I/O (c) Layout with embedded Network on Chip

Fig. 3. General Device Grid Examples

4.1 General Device Grid

Historical island-style FPGAs used column-based architectures (where all tiles in a column were of the same type) with I/Os located around the device perimeter as shown in Figure 3a. However modern FPGAs have more complex and non-uniform device grids, with I/Os often placed in columns instead of around the perimeter [57], and other large blocks such as hardened accelerators [9], embedded microprocessors [40, 121] and PLLs [13] located throughout the device grid. Other architectural ideas, such as embedding a Network-on-Chip (NoC) into the FPGA fabric (Figure 3c), may also result in non-columnar device grids [7, 107].

To this end we have extended VTR to support arbitrary device grids – there are no longer any location or size limitations for architectural blocks. For example Figure 3b shows a non-perimeter IO device with logic blocks, RAMs, I/Os and a hardened PCIE accelerator. Notably, the I/Os are not restricted to the device perimeter, and the accelerator block can be located at an arbitrary location spanning multiple rows and columns.

Large blocks also interact with the routing network architecture, requiring enhancements to the routing architecture generator (see Section 4.2.2). VTR 8's placement, routing and analysis routines (like timing and area estimation) all adapt automatically to the generated architecture.

4.2 Flexible Routing Architectures

4.2.1 Detailed Routing Architecture. To optimize for the characteristics and availability of different metal layers, modern FPGA routing architectures make use of wire types with different lengths, electrical characteristics and connectivity [92]. For instance long wires, which are usually implemented in the lower resistance upper metal layers, are used to quickly cross large distances but are relatively rare and usually have more restricted connectivity.

Previously, VTR supported only a limited number of switch blocks (Wilton [118], Subset/Planar/Disjoint [59] and Universal [20]) which defined a routing architecture's wire-to-wire connectivity pattern. These switch block patterns were developed with unit-length wires in mind, and do not describe different patterns between wire types, such as the hierarchical wire type connectivity used in commercial FPGA architectures [70, 71]. Similarly, VTR previously provided only limited control over the connectivity between block pins and different wire types.

In VTR 8 we have extended VTR's routing architectures generation capabilities to allow detailed control of both the switch block (wire-to-wire), connection block (pin-to-wire) and direct-connect (pin-to-pin) patterns. The switch block pattern is now highly customizable, allowing detailed control of how different wire types and individual tracks interconnect along their lengths. The connection block specification is also more flexible, allowing the

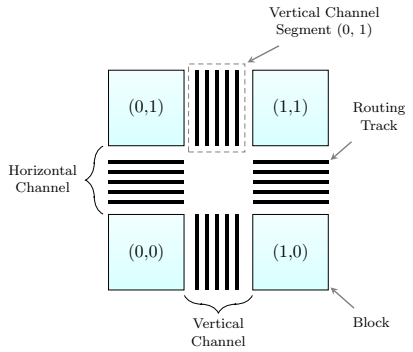


Fig. 4. Blocks and Routing Channels

connectivity between individual block pins and wire types to be specified. Finally, support for direct connections between block pins has also been extended to fully support connectivity between blocks of differing sizes (as shown in Figure 10).

These extensions give the architect more flexibility to define higher quality switch patterns which exploit different wire types [91], and more control over how the inter-block routing network connects to block pins. This also enables more accurate modelling of commercial routing architectures with hierarchical connectivity, as discussed in Section 5.

4.2.2 Routing Channels & Large blocks. VTR groups routing tracks which are logically adjacent into routing *channels* which are associated with a particular location in the FPGA device grid as shown in Figure 4. We refer to the wires above or to the right of a block as *channel segments*.

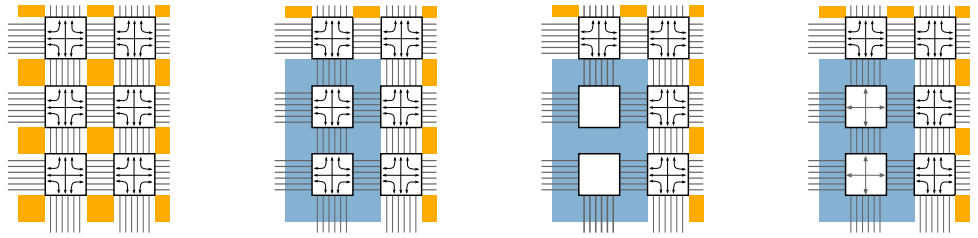
In addition to unit-sized logic blocks (Figure 5a), most heterogeneous FPGA architectures now include larger blocks (Figure 5b), such as large DSP blocks [9], large high-density RAM blocks (e.g. UltraRAM [120]), or other hardened accelerators (e.g. PCI-E, Ethernet [122]).

These types of blocks are often created following different design methodologies from the rest of the FPGA fabric. For instance, a DSP block or hardened PCI-E interface may be designed with a standard-cell-based methodology, while RAM blocks are usually built around a full-custom memory array. This makes integrating these blocks into the FPGA routing fabric challenging, particularly when the blocks are large enough to contain entire routing channels (both width and height > 1). There are several different approaches which can be taken, all of which are now supported in VTR 8.[6]

The simplest approach is to forbid the FPGA routing fabric from crossing the large block, as shown in Figure 5c. This allows the large block to be implemented independently from the FPGA fabric, but results in large routing blockages within the FPGA fabric.

Another approach is to reserve some of the metal layers above the large block, allowing FPGA fabric routing wires to cross over the block (Figure 5d). This has a smaller impact on the large block implementation (since only the reserved metal layer's can not be used by it), but requires no routing fabric switching to be pushed down into the block's active layers. Since signals can cross over the block this provides some routability improvement, but the lack of full switching capabilities between wires (switch-block) is still restrictive. This approach can also produce longer routing wires, since the wires on either side of the block are electrically shorted together, increasing their delay.

[6]Previous releases of VTR assumed all large blocks contained switch-blocks as in Figure 5b.



(a) Unit-sized blocks (b) Large block with full switch-block (c) Large block with no internal connections (d) Large block with shorted straight connections

Fig. 5. Switch-block connectivity within and around a unit and 2x3 sized blocks.

Table 1. Normalized Impact of 2x3 RAM block Internal Routing Connectivity on VTR benchmarks ($> 10K$ primitives)

	W_{min}	Routing Area ($1.3 \cdot W_{min}$)	Routed WL ($1.3 \cdot W_{min}$)	Crit. Path Delay ($1.3 \cdot W_{min}$)	Route Time ($1.3 \cdot W_{min}$)
none	1.00	1.00	1.00	1.00	1.00
short	0.86	0.86	1.06	1.02	1.01
full	0.86	0.92	1.00	1.00	0.94

Area is silicon area measured in Minimum Width Transistor Areas (MWTAs).

A more complex approach is to fully integrate the routing fabric into the large block (Figure 5b). This requires both reserving metal layers and pushing down routing fabric switches into the active layers of the large block's implementation. While this ensures maximum routing flexibility and performance it is highly disruptive to the large block's implementation, and may not be feasible in many cases.[7] As a result this approach likely only makes sense for extremely common blocks where the other approaches would cause too much disruption to the routing fabric.

VTR 8 supports all the above architectural options, allowing architects to specify how the routing network is implemented within and around different block types, enabling them to evaluate the different trade-offs between design time, area, delay, and routability.[8]

Table 1 shows the impact of these approaches on an architecture with length 4 wires and large 2x3 RAM blocks. Compared to the least flexible routing approach (`none`, Figure 5c), allowing wires to cross-over the block (`short`, Figure 5d) significantly improves routability, with minimum routable channel width (W_{min}) decreasing by 14%.[9] Interestingly, allowing full switch-blocks within the block (`full`, Figure 5b) does not improve minimum routable channel width compared to `short`, but increases area.[10] However `full` does improve routed wirelength, critical path delay, and route time compared to `short`.[11]

[7]For instance it may be difficult or impossible to embed routing switches into a RAM block's full-custom memory array.

[8]All of these architectural choices are used in various blocks of a variety of commerical FPGA architectures.

[9]Critical path delay and wirelength also increase moderately. As wires crossing the large block are shorted together in this architecture the additional loading will increase their delay. It may also increase wirelength, since they can no longer be used independently.

[10]The area increases due to the additional switch-block switches; however our area estimate neglects the likely significant area cost of disrupting the block's layout.

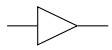
[11]While `full` achieves similar wirelength and critical path delay as `none` it does so at substantially smaller channel widths.

4.2.3 Non-configurable Switches. Traditionally, VTR has assumed that all switches in the routing network are *configurable* – meaning they can be configured to either connect or disconnect their input and output wires.

However some common circuit structures, such as inline non-configurable buffers along a wire, and electrical shorts between wires do not satisfy this assumption, as they always connect their input and output wires. To alleviate this, we added support for *non-configurable* switches which can not be disconnected or turned off, but must be used if their associated wiring is used.

This allows VTR to model structures such as wiring with inline buffers as shown in Figure 6a (which often appear in structures like clock networks), and non-rectilinear wiring (e.g. L, T or other shapes [103, 106]) as shown in Figure 6b which can be modelled as a combination of horizontal and vertical wire segments connected together with an electrical 'short'.

Section 9.3 describes the routing algorithm enhancements required to support non-configurable edges.



(a) Wire with inline buffer.

(b) Diagonal or 'L'-shaped wire modelled as rectilinear wires with an electrical short.

Fig. 6. Circuit Structures which can be modelled with non-configurable switches.

4.3 Area & Timing Modelling

VTR's modelling and analysis capabilities have also been extended.

4.3.1 Area Modelling. VTR now uses the COFFE area model [28], which provides more accurate area estimates on modern process technologies. In particular, it better captures the effects of N-well spacing and the combined effects of increasing transistor width and adding parallel diffusion regions.

4.3.2 Size Dependant Mux Delays. As VTR constructs an FPGA routing architecture it builds muxes with various numbers of inputs. While the VTR area model accounts for this, VTR 7 assumed a fixed delay regardless of mux size. VTR 8 now supports specifying a range of size-dependent mux delays, facilitating more accurate delay estimates.

4.3.3 Internal Primitive Timing Paths. Many components of an FPGA architecture are modelled as architectural primitives (e.g. LUTs, Flip-Flops). With increasing heterogeneity, a wider variety of primitives are used. Internally these components can be fairly complex and contain internal timing paths which will impact the timing behaviour of the overall system (e.g. DSP blocks, RAM blocks, hard memory controllers).

VTR 8 now supports modelling these internal timing paths, which can occur between a primitive's sequential input/output pins. It also supports multi-clock primitives where different ports are controlled by different clock domains.

Figure 7 illustrates these capabilities. The input ports `we1`, `addr1` and `data1` are sequential inputs clocked by `clk1`. These are connected to the sequential output `data2` (clocked by `clk2`) by an internal timing path. In contrast, the input `addr2` is a combinational input which connects to `data2`. VTR 8 supports any collection of combinational and sequential inputs (with multiple clocks) and internal timing edges/paths between them.

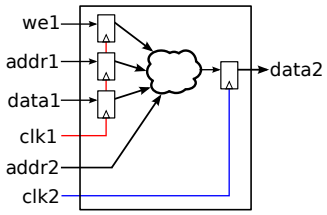


Fig. 7. Dual-Port RAM Block Internal Timing Path Example

4.3.4 Improved Delay Modelling. Traditionally, VTR has only supported the specification of maximum delay timing models. These models are used with Static Timing Analysis (STA) to guide implementation optimization, and report circuit critical paths and maximum operating frequencies.[12] However maximum-delay setup-time checks are not the only timing constraints which determine correct operation. It is also important to analyze minimum-delay timing for hold-time checks. To this end, VTR now supports the specification of minimum (in addition to maximum) timing information on all block delays, which is used for hold-time analysis (Section 10).

5 ARCHITECTURE ENHANCEMENTS CASE STUDY: STRATIX IV

The new architecture modelling capabilities described in Section 4 make it possible to model a much wider variety of FPGA architectures. To illustrate their utility, we now present improvements to the Stratix IV architecture capture used with the Titan benchmark suite [85].[13]

5.1 Grid Layout

Using the generalized grid support (Section 4.1), the device grid layout was adjusted to more accurately reflect real Stratix IV devices, as shown in Figure 8. PLLs are now placed within the I/O periphery and the I/Os are positioned so they can access vertical and horizontal routing regardless of where they are located. The M9K and M144K RAM blocks along with DSP blocks (which contain multipliers and accumulators) are arranged in columns. The remainder of the device is filled with Logic Array Blocks (LABs) consisting of Look-Up-Tables (LUTs), adders and Flip-Flops (FFs).

5.2 Routing Network

The routing network connectivity pattern was one of the largest approximations in the original Titan Stratix IV architecture capture from [85], since VTR 7 offered very limited control of the generated connectivity pattern.

5.2.1 Wire Connectivity, Switch Block & Routing Mux Sizes. VTR 7 supported only a small number of fixed switch blocks, with the Wilton switch block being the most commonly used. However the Wilton switch block was originally designed only considering wires of the same length. When used with multiple wirelengths the twisting pattern used by the Wilton switch block to improve routability means long wires rarely connect to each other. This defeats one of the primary purposes of having long wires in a routing architecture, which is to provide fast routes across long distances (accomplished by stringing together multiple long wires).

[12]The delay values for an architecture can be derived from the low-level architecture component implementations using tools like COFFE [28]

[13]Later Stratix series devices [71, 72] are similar to Stratix IV in the architecture dimensions considered here.

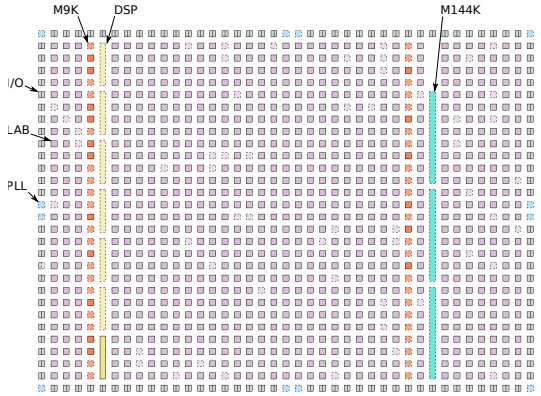


Fig. 8. Placement of the `leon2` benchmark with the updated grid layout and block type annotations.

By leveraging VTR 8's custom switch block support (Section 4.2.1) we can define a switch block which better captures the characteristics of the Stratix IV routing architecture. The characteristics modelled are summarized in Table 2. As in previous work [84, 85] we model the routing network as a mixture of 87% L4 and 13% L16 wires with a channel width of 300 tracks through-out the device.

Using VTR 8 we can correctly model the connectivity between the different wire types and between wires and block pins. In particular, the L16 wires are only accessible from the L4 wires and can neither drive nor be driven by block pins.[14] L16 wires also only connect to other wires via switch blocks at every 4th grid tile, and we correctly model this restriction.[15]

We also take care to match the size of the wire driver muxes, something which can be explicitly controlled with the custom switch block support in VTR 8. The L4 Driver Input Muxes (DIMs in Intel terminology) are sized to be 12:1. We distribute the inputs to these muxes such that 6 inputs come from block pins (determined by F_c), and the remaining 6 inputs come from a combination of other L4 and L16 wires. For the L16 DIMs Stratix IV uses larger muxes to increase the number of potential paths onto the long-wire network. Using VTR 8's custom switch-block support we were able to specify a routing architecture matching these characteristics.

Since the precise details of Stratix IV's switch pattern have never been disclosed, we experimentally evaluated a variety of routing patterns which matched the characteristics in Table 2 to find a pattern which performed well.[16]

5.2.2 Pin Locations. Stratix IV uses a three-sided routing architecture where each block can reach routing channels which are above, left or right of it [12]. By leveraging the enhanced pin location specifications, and improved control of routing connectivity around multi-height blocks (Section 4.2.2) this style of architecture can be accurately modelled.

In particular, for each logical pin on a block we create two physically equivalent pins: one on the top edge of the block, and the other on either the left or right side of the block. This allows each logical block pin to access both horizontal (above) and vertical (either left or

[14]Intuitively, this makes the rare L16 wires easier to use since the more numerous L4 wires serve as a feeder network for getting on to and off of the L16 wires.

[15]This is done in Stratix IV to reduce the expensive deep via stacks required to access transistors from the upper metal layers [70]

[16]Note these experiments, which evaluated which wires and which position along those wires were selected to drive each mux, were not possible to perform in VTR 7 due to limited user control over the generated switch pattern.

Table 2. Summary of Modelled Stratix IV Routing Architecture Characteristics

Metric	Expression	Value
Channel Width	W	300
L4 Wires	$0.8667 \cdot W$	260
L16 Wires	$0.1333 \cdot W$	40
L4 F_{cin}	$L4_{F_{cin}}$	0.055
L4 F_{cout}	$L4_{F_{cout}}$	0.075
L16 F_{cin}	$L16_{F_{cin}}$	0.000
L16 F_{cout}	$L16_{F_{cout}}$	0.000
L4 Driver Mux Size	$L4_{DIM}$	12:1
L16 Driver Mux Size	$L16_{DIM}$	40:1
L4 inter-switch-block distance	$L4_{ISBD}$	1
L16 inter-switch-block distance	$L16_{ISBD}$	4

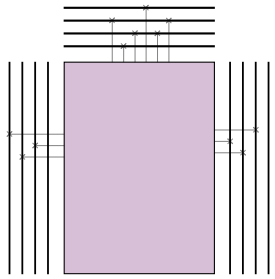


Fig. 9. Three sided architecture with each pin able to access one horizontal track, and one vertical track.

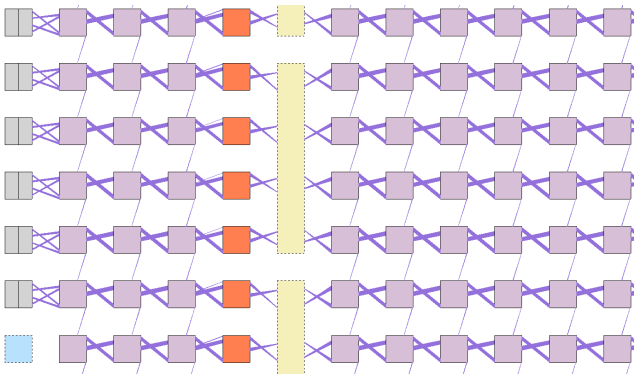


Fig. 10. Direct connections between adjacent blocks shown in purple, including horizontal 'direct-link' connections (between all blocks) and vertical carry-chain connections (between LABs). Block colors are the same as Figure 8.

right) routing wires, matching Stratix IV's capability (see Figure 9). By scaling the fraction of routing tracks to which each pin connects (F_c) appropriately we ensure the total number of wires connected to each logical pin remains constant.

5.2.3 Direct-Connect. We also extend the modelling of direct-link connections (which were previously modelled only between LABs) to include connections between blocks of differing types. In particular care was taken to ensure multi-height blocks like DSPs and M144K RAM blocks connect correctly to adjacent blocks which may have differing heights as shown in Figure 10.

We have also updated the architecture capture to include pack-patterns for all DSP block modes, ensuring that complex uses of the DSP block (like multiply-accumulate) are packed

efficiently and use the appropriate dedicated circuitry.

5.3 Timing Model

The timing model has been extended to include block internal timing paths (Section 4.3.3), such as those between registered RAM address ports and data output ports, and within registered DSP blocks.

A minimum delay timing model calibrated to match the component delays reported by Quartus STA is also included to enable hold-time analysis (Section 4.3.4).

5.4 Architecture Enhancements Comparison

Table 3 shows the impact of various architectural enhancements. In this experiment the Titan v1.1.0 benchmarks are run targeting both the original Stratix IV architecture capture (`A`) from [85] (which was describable in VTR 7), and several variants of the enhanced architecture with the custom switch-block and various DIM sizes (describable in VTR 8). The same version and configuration of VPR 8 is used in all cases.

Compared to the baseline architecture (`A`), the enhanced architecture (`E`) improves routing area (silicon area used for routing) by 11% and critical path delay by 8%. The critical path delay improvement is achieved in part by appropriately modelling the hierarchical long and short wire connectivity of Stratix IV. This ensures long wires always connect to each other, allowing critical signals to quickly traverse large distances by staying on the long-wire network.[17] The area reduction results from using smaller wire driver muxes than in the baseline architecture. Finally, the enhanced architecture is also easier to route, reducing route-time by 28%.

We also explored several other routing architecture variants, which tweak the size of the wire driver muxes. Note these experiments can not be performed in VTR 7, as the driver mux sizes can not be directly controlled. Compared to the baseline architecture (`A`), moving to the customized switch-block with hierarchical wire connectivity and fixed 12:1 driver muxes (`B`) reduces routing area by 13% and improves critical path delay by 5% while decreasing route time by 22%, at the cost of a small 3% increase in wirelength. Increasing the L16 driver muxes to 72:1 (`C`) makes it easier for critical connections to reach the fast long wires, further reducing critical path delay. While these muxes are much larger, there are relatively few L16 wires, so the overall area impact is less significant, but does reduce the area improvement to 8%. Increasing the much more common L4 driver muxes to 16:1 (`D`) reduces wirelength by 4% and critical path delay by 10% compared to the baseline, but at the cost of 9% higher area than (`C`) (1% higher area than the baseline). Interestingly, we found that decreasing the L16 driver mux size to 40:1 (like Stratix IV) had no significant impact on wirelength or delay, but reclaims some area compared to using 72:1 muxes.

6 LOGIC OPTIMIZATION & TECHNOLOGY MAPPING

VTR uses ABC to perform logic optimization and technology mapping. The version of ABC included with VTR 8 has been updated from an older 2012 version to a newer 2018 version [109]. We have also re-written our synthesis scripts to make use of ABC's new capabilities.

The first three rows of Table 4 show the impact of these changes on the large ($> 10K$ primitive) VTR benchmarks. Note that in these results only ABC is being modified; all other parts of the flow remain constant. Compared to using the old version of ABC (`2012`)

[17] This is not the case with the Wilton switch-block used in the legacy architecture, as it shuffles tracks in a manner which makes it unlikely the relatively few long wires will connect to other long wires.

Table 3. Geometric Mean Normalized Impact of Architecture Variants on the Titan Benchmarks

	Switch-block	$L4_{DIM}$	$L16_{DIM}$	Routing Area Per-Tile	Routed WL	Crit. Path Delay	Route Time
A	wilton			1.00	1.00	1.00	1.00
B	custom	12:1	12:1	0.87	1.03	0.95	0.78
C	custom	12:1	72:1	0.92	0.99	0.92	0.76
D	custom	16:1	72:1	1.01	0.96	0.90	0.79
E	custom	12:1	40:1	0.89	1.00	0.92	0.72

Area is silicon area measured in Minimum Width Transistor Areas (MWTAs).

and VTR's old synthesis script (`2012`), the new version of ABC (`2018`) produces smaller circuits (8% fewer primitives) resulting in a 10% reduction in routed wirelength, while ABC run-time decreases by 20%. Combining the new version of ABC (`2018`) with a new synthesis script (`2018`) further improves results, decreasing netlist primitives by 13% and logic depth by 10%; resulting in a 12% reduction in wirelength and 6% reduction in critical path delay. However this substantially increases ABC's run-time ($> 9\times$) compared to the 2012 ABC and script. This results in a small increase in total flow time since run-time is dominated by the search for W_{min}. However when running an implementation flow targetting a fixed channel width (and hence performs a single routing) ABC accounts for 35% of flow run-time on average, increasing overall flow time by $1.5\times$.[18]

6.1 Safe Multi-Clock Optimization

Multiple clock domains are a key characteristic of modern FPGA applications [35, 58]. One of the challenges with new versions of ABC is that all clock-related information is dropped during optimization.[19] This is dangerous, since it means sequential optimizations can occur across clock-domains, which can produce incorrect results. For instance, sequential elements with common fan-in but controlled by different clock domains could be incorrectly merged, or flip-flops controlled by one clock domain could be re-timed into logic related to another clock domain. Furthermore, special care is needed to safely transfer data across clock domains while avoiding synchronization and metastability issues [23]. Sequential optimizations may interfere with these transfers. Since VTR supports multi-clock circuits it is important to ensure synthesis is performed safely and all clock domain information is correctly maintained. To address this, VTR 8 adds support for several different approaches to ensure safe multi-clock logic optimization.

The most straightforward way to accomplish this is to hide a circuit's sequential elements from ABC by replacing them with black-box primitives which ABC does not optimize. However as shown in the fourth row of Table 4 black-boxing all sequential primitives (`all`) substantially degrades the quality of the resulting netlist compared to performing no black-boxing (`none`).

Instead VTR 8 defaults to an iterative black-boxing flow, where ABC is invoked multiple times, with only a single clock domain exposed as sequential elements; all sequential elements in other clock domains are black boxed. ABC therefore safely performs sequential optimizations on only one clock-domain at a time. As shown in the fifth row of Table 4 this approach (`iterative`) results in Quality of Results (QoR) similar to exposing sequential

[18] ABC can dominate run-time on some benchmarks, for instance accounting for 74% of overall run-time on `mcml`.

[19] Effectively ABC assumes all sequential circuit elements operate on a single shared clock domain. ABC also ignores their initial state.

Table 4. Normalized Impact of ABC Multiclock Flows on VTR benchmarks (> 10K primitives)

ABC Version	ABC Script	Multi-clock Black-box Flow	Primitive Blocks	Logic Depth	W_{min}	Routed WL ($1.3 \cdot W_{min}$)	Routed Crit. Path Delay ($1.3 \cdot W_{min}$)	ABC Time	Total Flow Time (find W_{min})	Total Flow Time ($1.3 \cdot W_{min}$)
2012	2012	none	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2018	2012	none	0.92	0.98	1.01	0.90	1.02	0.80	0.73	0.87
2018	2018	none	0.87	0.90	1.01	0.88	0.94	9.19	1.05	1.51
2018	2018	all	0.98	0.95	0.99	0.94	0.98	2.05	0.79	0.99
2018	2018	iterative	0.87	0.89	0.99	0.87	0.94	8.22	1.07	1.54

elements in all clock domains to ABC simultaneously (`none`) but ensures multi-clock circuits are implemented correctly.

7 PACKING ENHANCEMENTS

VTR supports a highly general packing algorithm based on greedy bottom-up seed-based clustering, which can target a wide variety of FPGA logic block and hard block architectures.

Seed-based clustering algorithms can achieve a very dense packing (high utilization per cluster) which helps minimize the number of clustered blocks required to implement a circuit. However, achieving high logic utilization has also been found to harm routability [34], and has been identified as a potential cause of routability issues with VTR 7 [85]. To this end we have focused on improving VPR's packing quality to produce more *natural* clusters, which better reflect the true relationships between netlist primitives. These more natural clusters are typically less dense and less interconnected which improves routability and overall implementation quality.

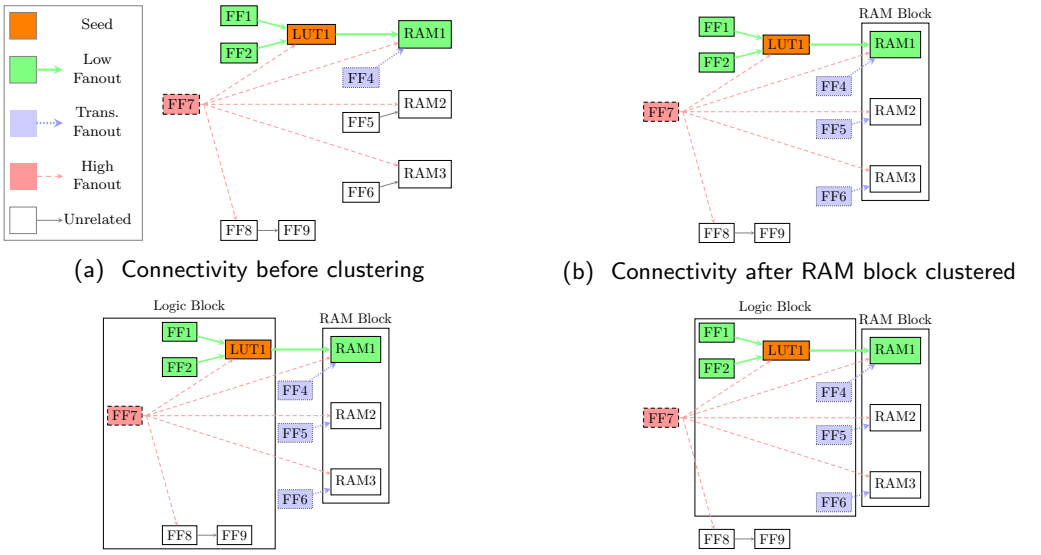
Algorithm 1 VPR 8 Packing Algorithm

Require: The target FPGA *architecture*, the *primitive_netlist* to be packed
Returns: The packed *clusters* implementing the *primitive_netlist*

```
1:  function VPR_PACK(architecture, primitive_netlist)
2:      unpacked_mols ← BUILD_MOLECULES(architecture, primitive_netlist)
3:      clusters ← ∅
4:      while unpacked_mols ≠ ∅ do
5:          seed_mol ← SELECT_SEED(unpacked_mols)
6:          cluster ← CREATE_CLUSTER(architecture, seed_mol)
7:          untried_mols ← unpacked_mols
8:          while SPACE_REMAINS(cluster) do
9:              next_mol ← PICK_NEXT_MOLECULE(cluster, untried_mols)
10:             if next_mol = ∅ then
11:                 break                                  ▷ Nothing productive to add
12:             if TRY_ADD_MOLECULE_TO_CLUSTER(cluster, next_mol) then
13:                 unpacked_mols ← unpacked_mols \ next_mol   ▷ Molecule is now packed
14:             untried_mols ← untried_mols \ next_mol         ▷ Do not retry molecule
15:         clusters ← clusters ∪ cluster                      ▷ Commit cluster
16:     return clusters
```

Algorithm 1 shows the key features of VPR 8's packing algorithm. Given a target architecture and primitive netlist to cluster, the packer groups primitives into *molecules* (Line 2), which are groups of primitives which must be packed together.[20] Next a seed molecule is

[20]An example of a multi-primitive molecule is a carry-chain. Each adder in the chain must be packed together to use a logic block's dedicated carry logic.



(a) Connectivity before clustering

(b) Connectivity after RAM block clustered

(c) After Packing Prioritizing Large Net Connectivity

(d) After Packing Prioritizing Transitive Connectivity

Fig. 11. Packing example with $\zeta = 4$.

selected (Line 5, described in Section 7.1) and used to create a new open cluster (Line 6). The cluster is then grown by selecting an unpacked molecule 'related' to the current cluster (Line 9, described in Section 7.2). The algorithm then attempts to add this candidate molecule into the cluster (Line 12). This continues until either the cluster is deemed to be sufficiently full (Line 8, described in Section 7.3), or no more 'related' molecules can be found (Line 10). At that point the current open cluster is closed (Line 15), and a new seed is selected from the remaining unpacked molecules (Line 5). This repeats until all molecules have been packed into clusters (Line 4). For more details on the generality of VPR's packing algorithm and how legality is ensured see [75, 79].

Some FPGA clustering approaches have bypassed bottom-up clustering [8, 24, 38, 114] using either partitioning or flat placement. However these techniques often relax legality constraints, are architecture specific, or still use bottom-up clustering as the final legalization step. The VPR packer faces unique constraints as it must support many different FPGA architectures which include different heterogeneous block types (e.g. DSP, RAM and custom blocks, in addition to logic blocks) with arbitrary internal connectivity (e.g. depopulated crossbars) and legality constraints (e.g. external/internal pin limits, special structures like carry-chains). Focusing on a bottom-up clustering approach makes it feasible to continue supporting this flexibility. Consistent with [101] we found that avoiding high-density clusters was important for improving routability and wirelength.

7.1 Where to Start?

Selecting a good seed primitive to start a cluster is an important consideration, since it impacts the order in which different parts of the netlist will be clustered. This is particularly important when transitive connectivity is used to determine relatedness.

This is illustrated in Figure 11, which shows the different types of connectivity the packer considers from the perspective of the seed primitive LUT1. Initially (Figure 11a) only FF4

is considered to be transitively related to LUT1 (since both LUT1 and FF4 connect to RAM1). However, after the RAM slices (RAM1, RAM2, RAM3) have been packed into a RAM block (Figure 11b), the packer gains additional information as it sees that FF5 and FF6 are also transitively related to LUT1 (via the RAM block). It is therefore beneficial to pack large blocks like RAMs and DSPs early during packing, so the packer has more complete transitive connectivity information.[21]

However, we found that VPR 7 did not always pack such blocks early. As a result we changed the seed selection criteria for timing-driven clustering to prefer this behaviour. This was accomplished by using a different criteria to rank potential seed molecules:

$$\begin{aligned} gain = {} & 0.5 \cdot norm\_input\_pins \\ & + 0.2 \cdot norm\_used\_input\_pins \\ & + 0.2 \cdot norm\_primitives\_in\_molecule \\ & + 0.1 \cdot primitive\_criticality \end{aligned} \tag{1}$$

Where *norm_input_pins*, *norm_used_input_pins*, *norm_primitives_in_molecule* and *primitive_criticality* are respectively: the number of input pins on the primitive (normalized to the primitive type with the most pins), the number of connected input pins (normalized to the primitive type's number of input pins), the number of primitives in the molecule (normalized to the largest molecule in the circuit), and primitive timing criticality.

The new gain function favours selecting primitives with a larger number of pins (such as DSPs and RAMs), ensuring they are clustered early in the packing process. In the case of a tie, the gain function favours blocks with more used inputs (i.e. which are connected in the circuit), and primitives which form parts of large multi-block molecules like carry-chains. Timing criticality is used as a final tie-breaker.

7.2 What to Pack Next?

After a seed molecule has been selected and a new cluster opened, the packer searches for unclustered molecules to add. This process is guided by *attraction functions* which estimate the gain (benefit) of adding an unclustered molecule to the current open cluster. We observed that VPR 7's attraction functions tend to produce somewhat unnatural clusters, with a mixture of related, loosely related, and unrelated logic. To address this we have made several enhancements to the attraction functions in VPR 8.

7.2.1 Prioritization of Connectivity Types. The packer considers three types of connectivity when selecting molecules to try adding to the current cluster.

Small net connectivity considers molecules connected to the current cluster via low-fanout nets, those with fanout below the high-fanout net threshold ζ (in VTR 7 $\zeta = 256$). In Figure 11a the connections between FF1, FF2, RAM1 and LUT1 are examples of this type of connectivity. Selecting candidate molecules connected by small nets is beneficial as it encourages nets to become completely absorbed within a cluster, reducing the number of inter-block connections which must be routed through the inter-block routing network [17].

Large net connectivity considers molecules connected to the current cluster via high-fanout nets (fanout $\geq \zeta$). In Figure 11a the connections from FF7 are examples of this type of connectivity. Large high-fanout nets are less beneficial for guiding packing than small nets, as they can rarely be well localized (e.g. absorbed into a single cluster).

[21] RAMs and DSPs usually have less packing flexibility and so benefit less from transitive connectivity information.

Transitive connectivity considers molecules which are connected to the current cluster indirectly through another molecule or cluster. In Figure 11b the connections from FF4, FF5, and FF6 are transitively connected to LUT1 through the RAM block. Transitive connectivity can be beneficial since it helps keep logic which is indirectly related together.[22]

VPR 7 prioritized small net connectivity first, followed by large net connectivity, and finally transitive connectivity. Since many molecules connect to at least one high-fanout net this meant VPR 7 only occasionally exploited transitive connectivity information. As a result, molecules which were only weakly related through high-fanout connectivity were often packed together – even when they were strongly related (e.g. via low-fanout connectivity) to other logic which had yet to be packed. The clusters which resulted often ended-up with split personalities: some portion of the logic was strongly related, but other parts were only weakly interconnected with the rest of the cluster (and may have had stronger connectivity to logic which ended up in other clusters).

This is illustrated in Figure 11c and Figure 11d. In Figure 11c, from the seed LUT1 small nets connectivity guides the packer to pack FF1 and FF2 into the cluster. Then, following large net connectivity, the only loosely related FF7, FF8 and FF9 are added to the cluster. In contrast Figure 11d follows the same small net connectivity to pack FF1 and FF2, but then follows transitive connectivity to select FF4, FF5 and FF6. This results in a more natural clustering.

In VPR 8 we have de-emphasized the importance of large net connectivity through two techniques. First, we prioritize transitive connectivity over large net connectivity. This ensures transitively related molecules are added to a cluster before those connected to high fanout nets. Second, we have lowered the threshold for high fanout nets (ζ) from 256 to 32 for logic blocks and 128 for all other block types.[23]

7.2.2 Unrelated Logic. As VPR packs a cluster it may be unable to find *any* unpacked molecules which are related to the current cluster. In VPR 7 the packer will continue to try filling up the cluster by finding unrelated molecules which will fit. While this minimizes the number of clusters created (by maximizing the utilization of each cluster) it can be detrimental to quality.

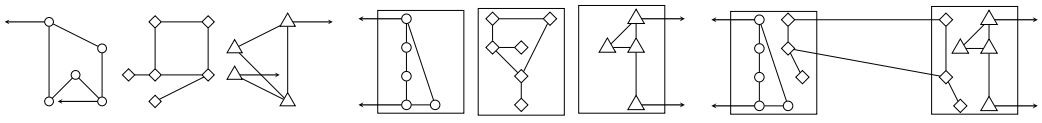
Consider the unclustered netlist in Figure 12a, which has a clear structure and can be naturally clustered as shown in Figure 12b by placing only related logic in each cluster. Allowing unrelated logic as in Figure 12c may result in fewer clusters but they are typically much more strongly interconnected; in this case leading to two highly connected clusters connected to other logic on opposite sides of the device.

Since the packer operates on a single open cluster at a time, 'unrelated' logic is only unrelated from the viewpoint of the current cluster and may be strongly related to other yet-to-be-packed parts of the netlist. As a result unrelated logic packing tends to scatter connected logic (which may have otherwise formed new more natural clusters) across many clusters. The resulting clusters are then coupled together through the unrelated logic making them more difficult to place and route. In VPR 8, unrelated logic packing is disabled unless too many clusters are produced to fit on a fixed-size device.

The impact of these changes on the `des90` benchmark is illustrated in Figures 13 and 14, which correspond to VTR 7-style and VTR 8-style packings respectively. In Figure 13 unrelated logic packing is enabled, producing fewer logic blocks which are more tightly

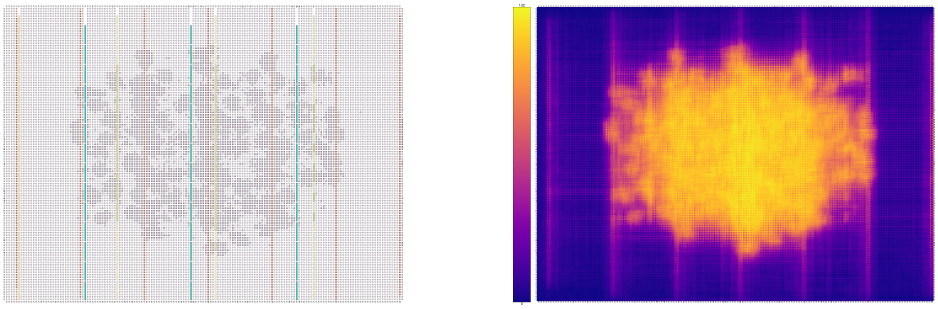
[22]For instance the FFs in a register bank may drive related logic but have no direct connectivity between themselves.

[23]A lower threshold treats more nets as high-fanout, further focusing the packer on small net connectivity.



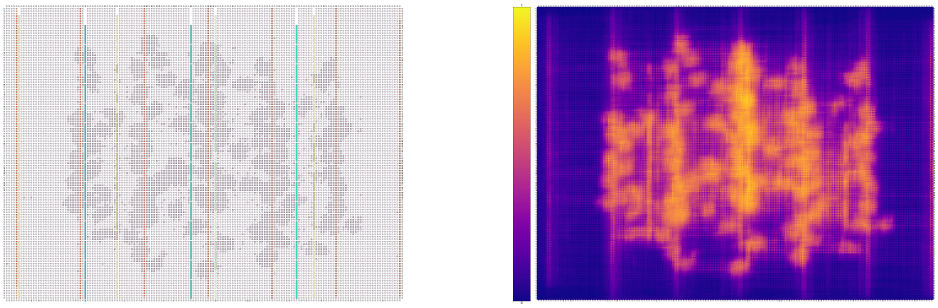
(a) Unclustered (b) Naturally Clustered (c) Clustered with unrelated logic

Fig. 12. Unrelated Clustering Example



(a) Placement (b) Routing utilization

Fig. 13. des90 high packing density (unroutable)



(a) Placement (b) Routing utilization

Fig. 14. des90 moderate packing density (routable)

coupled together. This coupling induces the placer to produce a tighter placement at the centre of the device to minimize wirelength (Figure 13a). However as Figure 13b shows, this causes significant routing congestion at the centre of the device, where 5066 channel segments[24] use $\geq 90\%$ of available wiring, resulting in an unroutable implementation. In contrast, with unrelated logic packing disabled (Figure 14a) the clusters are less interconnected and the placer is able to spatially localize the circuit's connectivity resulting in a more spread-out and natural placement. As Figure 14b shows, this significantly reduces routing congestion, and the design routes easily – with routing utilization $\geq 90\%$ in only two channel segments.

7.3 When is a Cluster Full?

Determining when a cluster should have no additional logic packed into it is also important for producing natural clusters.

When attempting to fill each cluster as full as possible (as VTR 7 does) some clusters may end up using all (or nearly all) of their external routing ports (i.e. cluster pins which

[24]A channel segment is all the wires in either a vertical or horizontal routing channel which overlap a particular grid tile, as illusrated in Figure 4.

connect to inter-block routing). While this high density packing may be beneficial from a utilization perspective (minimizing the number of logic blocks), it can make routing more difficult. Clusters with a large number of external routing connections will put additional stress on their adjacent routing channels – potentially inducing congestion.

7.3.1 Clustered Block Pin Utilization Targets. To avoid this behaviour we have modified the packer to consider the open cluster's pin utilization as part of the criteria for deciding when it should be closed. If the current open cluster exceeds its pin utilization targets it is considered full and is closed. These targets bias the packer away from creating difficult to route high pin utilization clusters. Importantly, these targets are treated as soft (rather than hard) constraints, which ensures netlist structure which may legitimately require more pins than the utilization target (e.g. large carry-chains) can still be packed.

VPR 8 now sets the default input pin utilization target of the logic block type to 80% of the available pins. All other blocks default to a 100% pin utilization target. This serves to guide the packer away from producing unnecessary high pin utilization logic blocks.

7.4 Adapting to Available Device Resources

While VPR is often allowed to dynamically re-size the device grid based on the benchmark circuit, it also supports targetting fixed-size devices. When targeting fixed-size devices it is possible for the packer to produce more blocks of a particular resource than are available on the device. In VTR 8 we have enhanced the packer to be resource aware and adapt to the resource mix of the targeted device. Together these changes ensure VTR produces a higher quality natural packing when there is sufficient space, and a denser clustering only if required.

7.4.1 Resource Balancing. The packer can optionally attempt to balance the resource utilization of different block types to better match the resources available on the device. For instance, if a netlist primitive can map to multiple block types (e.g. RAMs of two different sizes), the packer will attempt to produce a packing which balances the usage of the different block types in proportion to their quantity on the device.

On the Titan benchmarks (Section 12.2) this results in more RAMs being implemented as larger M144Ks (which are typically underutilized), instead of the more common M9Ks. This decreased the average device size needed to implement RAM limited circuits by 10%.

7.4.2 Adaptive Re-packing. The packer initially focuses on producing a natural packing, with unrelated logic packing disabled (Section 7.2.2) and reasonable pin utilization targets (Section 7.3.1). If this packing uses more resources than are available, the design is re-packed at a higher density with unrelated logic packing and resource balancing enabled.

7.5 Packing QoR Evaluation

Table 5 shows the cumulative impact of these modifications on the Titan benchmarks (Section 12.2),[25] where `baseline` corresponds to a VTR 7 style packing. Turning off unrelated logic packing (`no-unrel.`) had the largest impact reducing routed wirelength by 16%, decreasing router run-time by $> 2\times$, and allowing seven additional benchmarks to complete, at the (small) cost of 4% more logic blocks. Using the new seed formulation (`seed`) and an 80% input pin utilization target on logic blocks (`pin util.`) had limited effect. However, prioritizing transitive connectivity (`prioritize transitive`) further reduced (compared to

[25]The VTR benchmarks showed similar trends but smaller improvements.

Table 5. Cumulative Normalized Impact of Packing Changes on Titan benchmarks

	# Circuits Completed	LABs	DSPs	M9Ks	M144Ks	Routed WL	Crit. Path Delay	Pack Time	Place Time	Route Time	Total Time
baseline	13	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
no-unrel.	20	1.04	1.00	1.00	1.00	0.84	0.99	0.83	1.04	0.44	0.78
seed	20	1.05	1.00	1.00	1.00	0.84	0.98	0.86	1.04	0.42	0.78
pin util.	20	1.05	1.00	1.00	1.00	0.84	0.99	0.86	1.08	0.44	0.80
prioritize transitive	22	1.05	1.00	1.00	1.00	0.81	0.97	0.86	1.07	0.43	0.80

Normalized values are the geometric means over the common set of benchmark circuits which completed for all tool parameters

Table 6. Normalized Impact of Packer High Fanout Threshold on Titan benchmarks

	# Circuits Completed	LABs	DSPs	M9Ks	M144Ks	Routed WL	Crit. Path Delay	Pack Time	Place Time	Route Time	Total Time
256	22	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
64	22	1.01	1.00	1.00	1.08	0.96	0.99	0.97	0.98	0.90	0.97
48	23	1.01	1.00	1.14	1.21	0.98	0.98	0.88	0.85	0.85	0.89
32	23	1.03	1.01	1.19	1.22	0.97	0.99	0.93	0.97	0.85	0.97
128 LAB:64	22	1.01	1.00	1.00	1.00	0.97	0.99	0.87	0.89	0.89	0.90
128 LAB:32	23	1.03	1.00	1.00	1.00	0.95	0.98	0.83	0.87	0.85	0.87
128 LAB:16	23	1.07	1.00	1.00	1.00	0.94	0.99	0.80	0.91	0.78	0.88

Normalized values are the geometric means over the common set of benchmark circuits which completed for all tool parameters

the baseline) wirelength by 19% and improved critical path delay by 3%, while also enabling two more benchmarks to complete.

Table 6 shows the impact of modifying the packer's high-fanout net threshold (ζ) with other parameters set to those of the last row of Table 5. Decreasing the threshold from 256 to 64 improves wirelength by 4% and reduces route time by 10% while increasing the number of M144K RAM blocks by 8%. Further decreasing the threshold to 48 or 64 allows the largest Titan benchmark (`gaussianblur`) to complete, but further increases the number of RAM blocks. These results indicate that high-fanout net connectivity is useful for densely packing coarser blocks like RAMs and DSPs but can hurt the routability of the more fine-grained logic blocks. Decreasing the LAB high-fanout threshold to 32 while leaving the threshold at 128 for other block types (`128 LAB:32`) achieves the same routability improvement without increasing the number of DSP and RAM blocks. This validates the use of these values as the VTR 8 default.

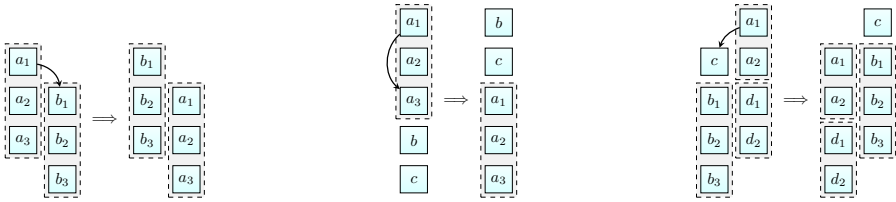
8 PLACEMENT ENHANCEMENTS

VPR's Simulated Annealing (SA) based placer iteratively makes a perturbation to the current placement (called a *move*) which is then evaluated and either accepted or rejected [15].

A desirable property for any move generator used in SA-based placement is that all possible configurations (i.e. placements) should be reachable through some (hopefully short) sequence of moves. Failure to ensure this means some parts of the placement solution space will either be impossible or very difficult for the placer to reach, potentially excluding higher quality placements. In VPR 8 we have made enhancements to the move generator to improve robustness and optimization quality.

8.1 Placement Macro Move Generator

VPR's placer supports *placement macros* which enforce relative placement constraints between specific blocks in the packed netlist. This is primarily motivated by architectural features such as fast dedicated routing for carry chains which require the connected blocks to have specific relative placements.



(a) Simple independent macro-macro swap.

(b) Self-overlapping macro swap (shift).

(c) Overlapping macro-macro swap. Moving macro a overlaps macro b, which when swapped also overlaps macro d.

Fig. 15. Macro swap examples.

As noted in [48], VPR 7's move generator had limited support for moves involving placement macros, and immediately aborted (gave up on) any moves involving two or more placement macros. This meant the placement of carry chains was not always well optimized as the placer required longer sequences of indirect moves to change the placement of two macros.[26]

In VPR 8 we have extended the placer's move generator to support moving two or more macros simultaneously. Table 7 shows the cumulative impact of supporting different types of increasingly complex moves involving multiple placement macros. Compared to the VPR 7-style baseline placer (`baseline`, which does not support moves involving more than one macro), supporting moves with two macros (`macro swap`, Figure 15a) improves routed wirelength by 2% and reduces route-time by 4%.[27] This more complex move generator increases placement time by only 1%.

We observed that despite this, many moves involving macros were still being aborted; particularly late during the anneal. This occurred because VPR's region limit[28] is reduced as placement progresses and as a result moves involving macros often either overlapped themselves or overlapped multiple nearby macros. For example, a move could propose shifting a carry chain by a distance shorter than the carry chain's length resulting in it overlapping its previous position (Figure 15b). Furthermore circuit structures like adder trees may cause carry chains to be placed close together, and as a result moving one carry chain may require shifting multiple carry chains to avoid overlaps (Figure 15c). Adding support for these types of overlapping macro moves (`overlapping macros` in Table 7) produces similar average QoR to `macro swap`, but enables the large carry-chain heavy `bitcoin_miner` design to successfully route.[29] On that design many proposed macro moves result in overlaps, so supporting such moves is key to allowing the placer to fine-tune the placement of carry chains.

8.2 Compressed Move Grid

When proposing moves, VPR 7 uses a region limit to control the distance between the involved blocks. Figure 16a shows the region limit around a selected block; the block can only be swapped with blocks within the region limit. As placement progresses the region

[26]This was particularly problematic on designs with many carry-chains, as proposed moves are more likely to involve multiple carry-chains in such cases. For instance in the Titan `bitcoin_miner` design 44% of logic blocks are part of a carry chain, and as a result > 49% of proposed moves were aborted.

[27]On `bitcion_miner` the move abort rate was reduced to 22.8%.

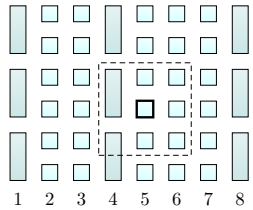
[28]The region limit controls how far apart the blocks involved in a move are allowed to be.

[29]The abort rate was further reduced to 0.6%.

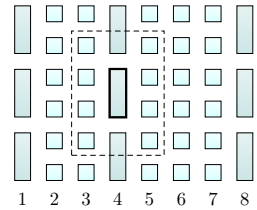
Table 7. Cummulative Normalized Impact of Placement Move Generation Changes on Titan benchmarks

	Completed Benchmarks	Placed WL	Placed CPD	Routed WL	Routed CPD (geomean)	Place Time	Route Time	Total Time
baseline	21	1.00	1.00	1.00	1.00	1.00	1.00	1.00
macro swap	21	0.97	0.98	0.99	1.00	1.01	0.96	0.98
overlapping macros	22	0.97	1.00	0.99	1.01	1.01	0.94	0.99
compressed move grid	22	0.92	0.98	0.94	1.02	1.01	0.86	0.97

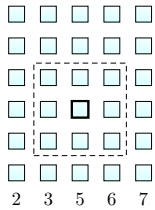
QoR and run-time metrics are normalized geomeans over mutually completed benchmarks



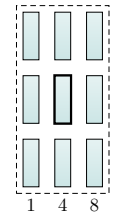
(a) VTR 7 move region limit for a common block.



(b) VTR 7 move region limit preventing a sparse block from moving between columns.



(c) VTR 8 move region limit for a block type in a compressed grid space. Common blocks can move across columns of other block types.



(d) VTR 8 move region limit for a sparse block in a compressed grid space. Sparse blocks can move between columns even at low range limits.

Fig. 16. Region Limit Examples

limit is decreased; focusing the placer on smaller, less disruptive changes which help fine-tune the placement.

One challenge with this approach is that it can prematurely lock down blocks which are relatively sparse within the FPGA device grid, as shown in Figure 16b. Once the region limit shrinks below the inter-column distance of a particular block type, those blocks become stuck within their current column and will never be able to move to another column. If this happens too early in the anneal such blocks may be prematurely locked down in sub-optimal positions.

To counteract this, VTR 8 does not apply the region limit directly to the FPGA grid. Instead, it constructs a compressed grid space for each block type to which the region limit is applied. This ensures blocks of a particular type in the FPGA grid which are logically adjacent to each other are considered as such during move generation. As shown in Figures 16c and 16d this prevents the range limit from artificially restricting the movement of some block types. Using these compressed grid spaces also ensures VPR can always quickly find adjacent blocks of the same type.[30]

The results of this change are listed as `compressed move grid` in Table 7. Using compressed move grids further improves routed wirelength and route-time by 6% and 14%

[30]Previously in VTR 7 the placer would abort some moves if it was unable to quickly find another block of the correct type.

respectively. While routed critical path is shown to increase by 2%, this excludes the improvements on benchmarks like `bitcoin_miner` which did not complete in the baseline. However the post-placement estimated critical path delay (which includes all benchmarks) improves by 2%.

9 ROUTING ENHANCEMENTS

VPR's routing algorithm is based upon the Pathfinder negotiated congestion algorithm [80], which repeatedly rips-up and re-routes nets while modifying routing resource costs based on occupancy and previous congestion. Routing consumes a large portion of the CAD flow run-time [85] and can scale poorly to very large designs due to the difficulty of resolving congestion and routing high-fanout nets. As a result we have implemented a number of enhancements which both improve the quality and run-time of the VPR 8 router.

VPR 8's netlist routing algorithm, AIR [86], is outlined in Algorithm 2. The algorithm operates over multiple *routing iterations* (Line 4). During each iteration, connections are routed between each net's driver and sinks (Lines 6 and 7). Importantly, each connection is allowed to use routing resources already used by other nets. Such overused routing resources are said to be *congested*. Once all nets have been routed (Line 5), the routing resource costs are selectively increased (Line 16) with the aim of reducing congestion during subsequent routing iterations. This process repeats until a legal routing is found (Lines 8 and 11), or the design is deemed unroutable by either: prediction (Line 14) or hitting the maximum iteration limit (Line 4). Finally, the best legal routing found (if any) is returned (Line 20).

We describe enhancements to the core routing algorithm in Sections 9.1 and 9.2, and extensions to support non-configurable switches in Section 9.3. Improvements to minimum channel width search are described in Section 9.4, and Section 9.5 evaluates the enhanced router.

9.1 A Fast but Lazy Router

9.1.1 Incremental Routing. In a traditional pathfinder-based routing algorithm all nets are ripped-up and re-routed each routing iteration. However in practise many nets (or portions of nets) may have been legally routed. In such cases it is not strictly necessary to re-route previously legal connections.

In VTR 8 net connections are routed incrementally to avoid this redundant work. First, a net's route-tree from the previous iteration is walked from the root to identify any congested sub-trees as shown in Figure 17a. The illegal sub-trees are then pruned away leaving only the legal portion of the previous routing (Algorithm 2 Line 18) as shown in Figure 17b. Next, the net sinks which were pruned are re-routed during the next routing iteration (Algorithm 2 Line 6), using the existing legal routing as the starting point.

The work in [115] proposed decomposing nets into independent connections, which allowed the creation of a routing algorithm with reduced run-time. Unlike [115] our approach does not strictly decompose the routing problem into connections and maintains its natural net-based structure. This allows partial routing from previous connections to be re-used which reduces the amount of graph exploration required and enables further net-based optimizations such as high-fanout routing (Section 9.1.2).

While incremental routing is primarily a run-time optimization it is important to consider whether failing to rip-up legal connections can have any impact on QoR. In particular, since Pathfinder uses a present congestion cost factor, not ripping up all connections may result in some timing critical nets taking more indirect routes to avoid present congestion. During experiments we found that in some instances this could degrade critical path delay. To

Algorithm 2 VPR 8 Netlist Routing

Require: The *nets* to route, α the maximum number of routing convergences
Returns: The *best* routing found

```
1:  function VPR_ROUTE(nets, α)
2:      best ← ∅, current ← ∅                    ▷ Best and current route trees for each net
3:      convergences ← 0                          ▷ How many times a legal routing has been found
4:      for iter ∈ 1 ... max_iters do
5:          for net ∈ nets do
6:              for sink ∈ UNROUTED_SINKS(net, current[net]) do    ▷ Incrementally route
7:                  current[net] ← VPR_ROUTE_CONNECTION(net, current[net], sink)
8:          if IS_LEGAL(current) then
9:              best ← BEST_ROUTING(best, current)                 ▷ Keep best routing so far
10:             convergences ← convergences + 1
11:             if convergences = α then                           ▷ Convergence limit reached
12:                 break
13:             RESET_PRES_FAC()          ▷ Reduce present congestion cost to focus on delay
14:         if PREDICT_UNROUTABLE() then                                        ▷ Early abort
15:             break
16:         UPDATE_COSTS()                                          ▷ Update pathfinder costs
17:         for net ∈ nets do                          ▷ Prepare for incremental re-routing
18:             RIPUP_ILLEGAL_CONNECTIONS(current[net])
19:             RIPUP_DELAY_DEGRADED_CONNECTIONS(current[net])
20:     return best
```

Algorithm 3 VPR 8 Connection Routing

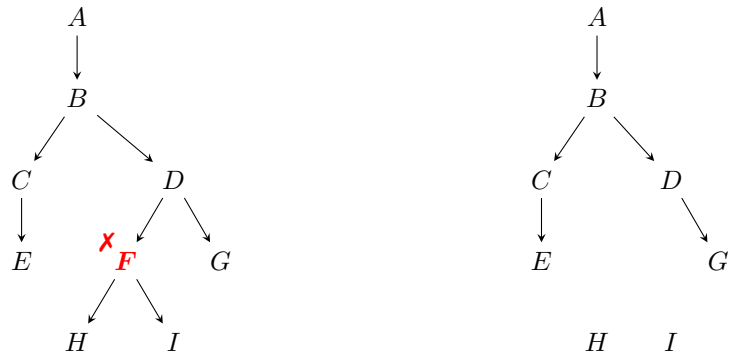
Require: The *net* to route, its existing *route_tree*, the target *sink* to be added
Returns: The updated *route_tree* with a branch connecting to *sink*

```
1:  function VPR_ROUTE_CONNECTION(net, route_tree, sink)
2:      path ← ∅
3:      if FANOUT(net) ≥ β and CRITICALITY(sink) < γ then
4:          heap ← INITIALIZE_NEARBY_ROUTING(route_tree, sink)      ▷ High-fanout opt.
5:          path ← FIND_PATH_FROM_HEAP(heap, sink)
6:      if path = ∅ then
7:          heap ← INITIALIZE_FULL_ROUTE_TREE(route_tree)
8:          path ← FIND_PATH_FROM_HEAP(heap, sink)
9:      UPDATE_ROUTE_TREE(route_tree, path)
10:     return route_tree
```

alleviate this behaviour we also perform delay-based rip-up (Algorithm 2 Line 19), which will force critical connections whose delay has degraded (compared to previous results) to be re-routed even if they have a legal routing.

While this technique can save significant work for large nets, on small low-fanout nets the work performed pruning the route tree is comparable to the work required to re-route the net. As a result we apply incremental net re-routing only for nets beyond a certain fanout threshold.

Tables 8 and 9 compare incremental routing at various fanout thresholds to the `Baseline` method (where nets are always ripped up) on the VTR and Titan benchmarks respectively,



(a) Route tree from A to $\{E, H, I, G\}$. Node F is illegal (also used by another net).

(b) Pruned legal partial route tree. Only sinks H and I must be re-routed. Any of $\{A, B, C, D, E, G\}$ could be new branch points.

Fig. 17. Route tree pruning example.

Table 8. Normalized Impact of Incremental Routing on VTR benchmarks ($>$ 10K Primitives)

	W_{min}	Routed WL ($1.3 \cdot W_{min}$)	Crit. Path Delay ($1.3 \cdot W_{min}$)	Route Time (find W_{min})	Route Time ($1.3 \cdot W_{min}$)	Peak Memory	Total Flow Time
Baseline	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8	1.01	0.99	1.01	0.63	0.77	1.01	0.72
16	1.01	0.98	1.03	0.66	0.77	1.01	0.76
32	1.02	0.99	1.00	0.75	0.81	1.01	0.84
64	1.01	0.99	1.01	0.82	0.90	1.01	0.86
128	1.00	0.99	1.01	0.82	0.97	1.00	0.90

Table 9. Normalized Impact of Incremental Routing on Titan benchmarks

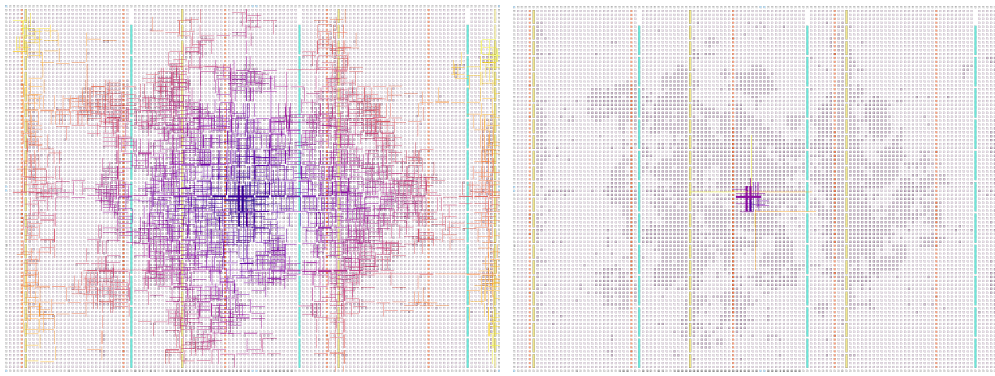
	Routed WL	Crit. Path Delay	Route Time	Peak Memory	Total Flow Time
Baseline	1.00	1.00	1.00	1.00	1.00
8	0.98	1.00	0.85	1.00	0.93
16	0.99	1.00	0.81	1.00	0.87
32	0.99	1.00	0.85	1.00	0.87
64	1.00	1.00	0.85	1.00	0.88
128	1.00	1.00	0.87	1.00	0.91

Excluding: `directrf`, `gaussianblur`, `dart`

with all other routing optimizations enabled. The results show that incremental routing has effectively no impact on quality of results (minimum routable channel width, wirelength, or critical path delay), while reducing the run-time of the router. For the VTR benchmarks (Table 8) and Titan benchmarks (Table 9) fixed channel width route-time is minimized at a fanout threshold of 16, reducing average route-time by 23% and 19% respectively. Importantly, the impact of incremental routing is more significant on larger circuits (run-time reduction of 56% on `directrf`), which shows it improves scalability.

9.1.2 High-Fanout Nets. VPR routes each net one connection at a time (Algorithm 2 Line 7) using Algorithm 3. To avoid wasting large amounts of wiring, any existing routing (from a net's previously routed connections) is added to the heap with zero cost, allowing it to be re-used as a branch-point for subsequent connections (Algorithm 3 Line 7).

Most designs have a relatively small number of high-fanout nets which span a large portion of the device. On a subset of the Titan benchmarks we found VPR spent 12-34% run-time



(a) Traditional approach (b) Spatial approach

Fig. 18. Routing resources explored (colored by cost to reach the current target sink) while routing a high-fanout net (~ 3500 terminals) on the `neuron` benchmark circuit. The target sink is located near the centre of the device.

routing these few high-fanout nets.

For a net with k sinks, putting the entire route tree into the heap means the time complexity to route the net grows quickly. For a single sink, pushing the net's $O(k)$ route tree into the heap has a time complexity[31] of:

$$O(k \log k). \tag{2}$$

Since this is done for each of the k sinks, the overall time complexity to route a net grows as:

$$O(k^2 \log k). \tag{3}$$

The $O(k^2)$ component of Equation (3) is problematic, as it leads to very high route-time for high-fanout nets.

To avoid this VTR 8 uses an approach based on [108]. Rather than putting the entire route-tree into the heap, only routing resources which are spatially near the target sink are added (Algorithm 3 Line 4). Since the number of such resources is bounded by a small constant this reduces the $O(k \log k)$ component of Equation (3) to $O(1)$, and the complexity of routing a net falls to:

$$O(k), \tag{4}$$

which is much more scalable.

Figure 18 illustrates the differences between these approaches, by showing the routing resources examined by the router when routing to a particular high fanout net sink located near the center of the device. Figure 18a shows the traditional approach looks at a large number of routing resources (i.e. considers the entirety of the large route tree), many of which will not lead to the target sink (since they are far from it). In contrast, Figure 18b shows the spatial approach is much more efficient, with the router only examining a few routing resources which are near the target sink.

However, we found that on some FPGA architectures this technique was not robust, and no path could be found from the spatially nearby previous routing and the target sink. This can occur in routing architectures which have sparse connectivity to certain routing resources such as long wires. In such cases we fall back to placing the full route tree onto the

[31] Assuming a binary heap with $O(\log k)$ insertions.

Table 10. Normalized Impact of High Fanout Routing on Titan benchmarks

	Routed WL	Crit. Path Delay	Route Time
off	1.00	1.00	1.00
$\beta = 32$ $\gamma = 1.0$	1.00	1.03	0.84
$\beta = 64$ $\gamma = 1.0$	1.00	1.03	0.86
$\beta = 64$ $\gamma = 0.9$	1.00	1.00	0.77

Table 11. Normalized Impact of OPIN Lockdown on Titan benchmarks

	Routed WL	Crit. Path Delay	Route Time	Peak Memory
on	1.00	1.00	1.00	1.00
off	0.98	1.01	0.94	1.00

heap (Algorithm 3 Lines 6 to 8). Since such instances are rare, this maintains the speed-up of spatially aware high-fanout routing while ensuring robustness.

Furthermore, we observed that this high-fanout routing technique can also degrade timing performance, since high-fanout timing-critical connections will have less flexibility to find potentially faster routes. To alleviate this we route all timing critical high-fanout connections[32] using the full previous route tree (Algorithm 3 Line 3).

While these optimizations had no significant effect on the VTR benchmarks, they do impact the larger Titan benchmarks. Table 10 show the results for several fanout (β) and criticality (γ) thresholds (Algorithm 3 Line 3), with all other routing optimizations enabled. While high-fanout routing reduces run-time the best result occurs with a fanout threshold of 64 and criticality threshold of 0.9, where route time is reduced by 23% with no impact on wirelength or critical path delay. Notably, setting an active criticality threshold ($\gamma < 1$) improves route time since it likely avoids having to rip-up and re-route otherwise delay sub-optimal high-fanout connections.

9.1.3 Output-pin Lockdown. To facilitate faster routing convergence, the VPR 7 router locks-down output pins after a fixed number of routing iterations, with the aim of preventing signals from oscillating between different logically equivalent output pins as congestion is resolved. As shown in Table 11 (with all other routing optimizations enabled), disabling this feature reduces both wirelength and route-time on the Titan benchmarks. VPR 8's incremental re-routing enhancement (Section 9.1.1) avoids unnecessarily ripping up and re-routing uncongested output pins, naturally avoiding the scenario output pin lockdown tried to address in VPR 7. As a result output pin lockdown has been removed in VPR 8.

9.2 Improving Quality & Robustness

9.2.1 Router Lookahead. VPR's connection-based router makes use of a predictive lookahead, to estimates the cost (delay and congestion) of reaching the target sink through the node being expanded (Algorithm 5 Lines 10 and 11). This lookahead is used to quickly guide the router to find a low cost path to the target.[33] The lookahead used by prior versions of VPR, which we call the *classic* router lookahead, makes a variety of simplifying assumptions which may not hold true on modern FPGA architectures. In particular, it assumes different wire types do not interconnect, and all wire types connect to block pins. As a result, the classic lookahead can mislead the router on modern architectures (where its assumptions typically do not hold), harming delay and wirelength.

[32]For the first routing iteration we treat all connections as timing critical. For later iterations the criticality is determined based on the delays of the previous routing iteration.

[33]This is an approximate variant of the A* algorithm [47].

Table 12. Normalized Impact of Router Lookahead and Base Costs on Titan benchmarks

	Routed WL	Crit. Path Delay	Route Time	Peak Memory
classic	1.00	1.00	1.00	1.00
map	0.98	0.91	5.94	1.00
classic_length	0.93	0.99	0.76	1.00
map_length	0.92	0.91	0.89	1.00

Values are normalized geomeans over mutually completed benchmarks

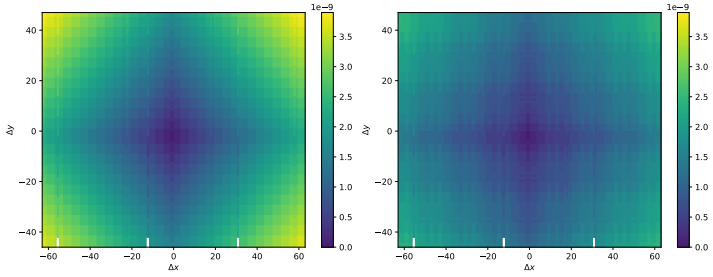
As a result VPR 8 includes a new lookahead based on an enhanced version of [92]. Unlike the classic lookahead, which made fixed assumptions about the FPGA routing network structure, the new lookahead adapts to the routing architecture used. This is accomplished by profiling the routing network with sample routes to build a 'map' of the delay and congestion costs through the routing network. To keep the creation time and memory footprint reasonable, the map lookahead makes the common assumption that the FPGA routing network is translationally invariant, and only differences in position need to be considered. The map lookahead stores a different delay/congestion cost map for each source wire type (e.g. wire length) and orientation (vertical or horizontal).

Table 12 shows the impact of the different lookaheads on the Titan benchmarks, with all other router optimizations enabled. First, compared to the VPR 7-style lookahead (`classic`), the new map lookahead (`map`) achieves much better result quality; reducing critical path delay by 9% and wirelength by 2%, but at the cost of increasing route-time by nearly 6× (the high run-time will be alleviated by adjusting the base costs as described in Section 9.2.2). The map lookahead improves critical path delay since it understands the hierarchical structure of Stratix IV's routing network, where the high-speed long wire sub-network is only accessible from a subset of the more plentiful short wires (Section 5.2.1). This is illustrated in Figure 19 which shows the delay estimates for various distances starting from a short wire for both lookaheads. In Figure 19a the classic lookahead assumes all connections use the same type of short wire, leading to delay rapidly increasing with distance. In contrast, Figure 19b shows the map lookahead's delay estimate, which increases much slower, particularly for longer distances. The map lookahead understands the interconnections between the different wire types, so it knows it is possible to get onto the faster long wire network even when starting from a short wire. As a result its delay estimate increases slower at longer distances since it knows the faster long wire network can be used.

These differences guide the router to make different choices for timing critical long distance connections. With the classic lookahead, the router immediately starts driving towards the target using the short-wire network, since it does not understand faster paths may exist. It will only use the long wire network if it happens to encounter it during its directed exploration. The map lookahead instead guides the router to search the short-wire network more thoroughly to find a way onto the faster long wire network. Getting onto the long wire network early significantly improves the delay of long distance connections.

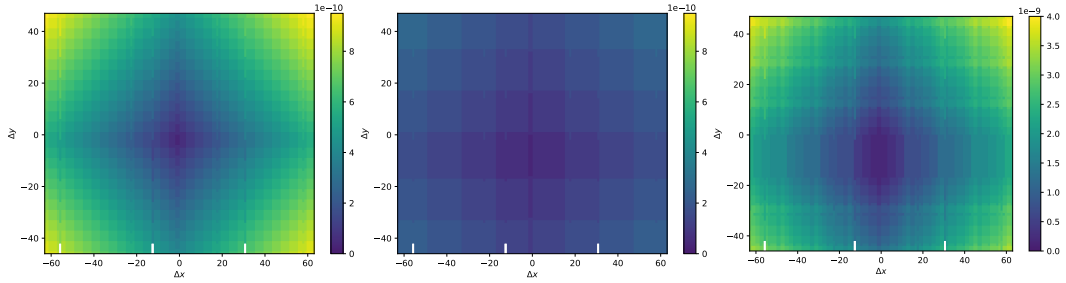
9.2.2 Routing Resource Base Costs. The lookaheads also provide congestion/resource cost estimates to guide the router's search process. Figure 20 shows the congestion cost estimates. Figures 20a and 20b show the classic lookahead's congestion cost estimates when starting from short or long wires. Interestingly, this shows it is much cheaper to travel an equivalent distance using the long rather than short wires – even though the long wires are much larger and rarer routing resources.[34] This cost difference biases the router to prefer using the long

[34]This is derived from the original VPR formulation [17], which used a single base cost for all wire lengths.



(a) Classic L4 (b) Map L4

Fig. 19. Lookahead Delay Estimates



(a) Classic L4 (b) Classic L16 (c) Map Length L16

Fig. 20. Lookahead Congestion Estimates

wire network even for non-timing-critical connections.

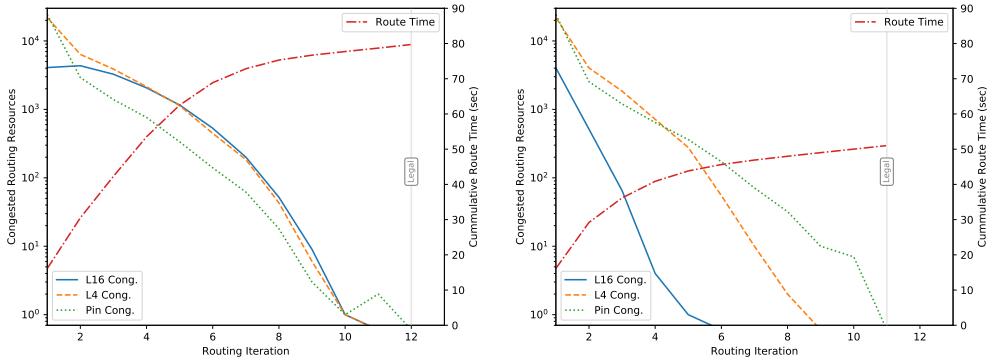
The original map lookahead suffers from the same bias. Since the map lookahead also has a stronger preference for using long wires for timing-critical connections this led to significant congestion in the long wire network. While this congestion would eventually be resolved through congestion negotiation, that is a very slow process, requiring connections to be repeatedly ripped-up and re-routed over many routing iterations.

To address this issue we adjusted the wire base costs to be proportional to each wire's length. The resulting congestion cost estimates for long wires with the map lookahead are shown in Figure 20c. These costs make long wires more expensive than short wires, particularly when using a long wire to travel distances shorter than their length.[35] This guides short distance and non-timing-critical connections to use the more plentiful short wires. As a result, wirelength improves, and run-time decreases (as congestion is resolved quickly).

This is illustrated in Figure 21 which compares how base costs effect congestion resolution and route-time. For non-length-scaled base costs (Figure 21a) both the L4 and L16 wires are initially heavily congested. However the L16 wires are proportionally much more congested than the L4 wires.[36] Congestion in both wire lengths then resolves very slowly over many routing iterations. In contrast, for length-scaled base costs (Figure 21b) congestion in both wire lengths resolves quickly. This is particularly true of the L16s, which are congestion free much earlier (6^{th} iteration) than with unit base costs (11^{th} iteration). As a result the

[35]For instance, with the previous uniform base costs using an L16 wire to move 8 units was half the cost of using 2 L4 wires. Using length-scaled base costs the L16 wire would be twice as expensive as using 2 L4s, which is intuitively consistent as half the L16 wire would be unused.

[36]There are $\sim 26\times$ fewer L16 than L4 wires (Table 2), but initially only $5\times$ fewer congested L16s in Figure 21.



(a) Unit base costs. (b) Length-scaled base costs.

Fig. 21. Comparison of routing congestion by resource type and router run-time for different base costs on the `neuron` benchmark circuit with the map lookahead.

router converges in fewer routing iterations, and requires significantly less run-time (80 vs 50 seconds).[37]

The impact of using length-scaled base costs with the two lookaheads is also shown in Table 12. With length-scaled base costs the map lookahead (`map_length`) significantly improves route-time, so it is 11% *faster* than the classic lookahead. They also improve the classic lookahead's run-time (`classic_length`), but the relative improvement is smaller. For both lookaheads the length-scaled base costs reduce routed wirelength by 7-8%.[38]

9.2.3 Multi-convergence Routing. During negotiated congestion routing, the router attempts to resolve congestion while minimizing the impact on critical path delay. However in highly congested designs congestion costs can end-up dominating timing costs, and critical connections may be detoured degrading the critical path delay. As a result the achieved critical path delay can be somewhat chaotic in the presence of significant routing congestion.

Multi-convergence routing aims to address this by continuing to improve the routing of timing critical connections *after* the first legal routing solution is found. In particular, the router will continue re-routing timing critical connections with the aim of improving their delay. This allows the router to fix up timing issues it neglected while focusing on resolving routing congestion. When the routing next converges to a legal congestion-free solution the routing with the best QoR is kept.

The pseudo-code for multi-convergence routing is shown in Algorithm 2. When a legal routing is found (Algorithm 2 Line 8), the best routing is updated[39] (Algorithm 2 Line 9). In preparation for the rip-up and re-routing of delay sub-optimal connections the present congestion cost factor ($pres_{fac}$) is reset to its initial value (default zero) (Algorithm 2 Line 13). This allows timing critical connections to focus on finding faster routes instead of avoiding congestion. Incremental re-routing (Section 9.1.1) then rips-up delay sub-optimal connections (Algorithm 2 Line 19). It is worth noting that once re-routing is 'kicked-off' in

[37]Re-routing in the presence of congestion requires the router to repeatedly search more and more of the RR graph to find uncongested paths. As a result, the run-time benefits of reducing congestion tend to improve with the larger RR graphs needed for bigger designs.

[38]This likely indicates long wires were previously sub-optimally used by connections traversing short distances. This was also independently identified and fixed by [115].

[39]Provided the new routing is better, defined as having lower critical path delay, with wirelength used as a tie-breaker.

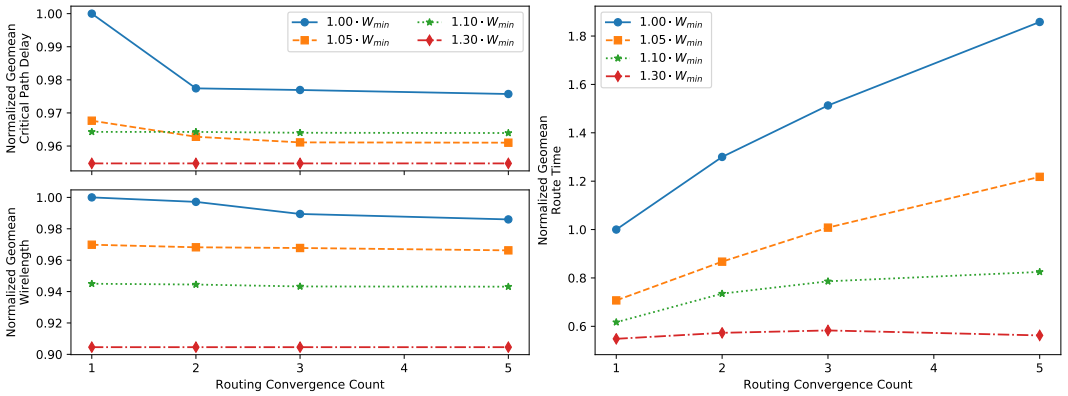


Fig. 22. QoR Impact of multiple routing convergence on the VTR Benchmarks ($> 10K$ primitives)

this manner any resulting congestion is naturally handled by incremental re-routing, and less critical (but newly congested) connections will be detoured away to resolve congestion.

Figure 22 shows the impact of multi-convergent routing on the large VTR benchmarks at various levels of routing stress, and with all other router optimizations enabled. We can make several interesting observations.

Firstly, independent of multi-convergence routing, increasing channel width improves both delay and wirelength, and reduces route-time. The additional routing resources mean the router does not need to detour as drastically to resolve congestion.

Secondly, multi-convergence routing improves critical path delay and wirelength in high stress settings at or near the minimum routable channel width. For instance, compared to only a single convergence, allowing two convergences reduces critical path delay at minimum channel width by 2.3%. However these gains diminish as channel width increases. Allowing more than two routing convergences offers minimal quality benefit.

Thirdly, multi-convergence routing is run-time efficient, with subsequent convergences increasing run-time by far less time than the initial convergence (which routed the entire netlist). For instance at minimum channel width the second convergence only increased overall route-time by 30%. The run-time overhead of multi-convergence routing also decreases in less stressful routing conditions (where it offers less benefit). The connection-based re-route and high fanout optimizations greatly reduce work when a routing is almost legal, which keeps the run-time overhead for multi-convergent routing low.

It is interesting to note that multi-convergent routing with delay-based rip-up accomplishes many of the same goals as the delay-targeted routing approach proposed in [96]. In particular, reducing the often chaotic impact of routing congestion on critical path delay at narrow channel widths. Multi-convergent routing should be more run-time efficient as it only re-routes the relevant connections in the netlist. In contrast delay-targeted routing requires the full netlist to be re-routed multiple times in search of an appropriate delay target. Finally, multi-convergent routing naturally extends to multi-clock designs where there is no longer a single delay target.

9.2.4 Adapting to Congestion.

Dynamic Router Bounding Boxes. To reduce router run-time the VPR router has historically restricted each net to route within a fixed region derived from the bounding box of the net's

Table 13. Normalized Impact of Dynamic Router Bounding Boxes on VTR benchmarks ($> 10K$ primitives)

	W_{min}	Route Time (find W_{min})	Routed WL (Fixed W)	Crit. Path Delay (Fixed W)	Route Time (Fixed W)
static	1.00	1.00	1.00	1.00	1.00
dynamic	0.98	1.11	1.00	1.00	0.98

Table 14. Normalized Impact of Congestion Mode Threshold on VTR benchmarks ($> 10K$)

	W_{min}	Routed WL ($1.3 \cdot W_{min}$)	Crit. Path Delay ($1.3 \cdot W_{min}$)	Route Time (find W_{min})	Route Time ($1.3 \cdot W_{min}$)	VPR Memory	Total Flow Time
1.0	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.8	0.99	1.01	1.00	1.08	1.00	1.00	1.01
0.5	0.98	1.01	1.00	1.57	1.00	1.00	1.23

driver and sink locations.[40] While this limiting of the router's search space saves run-time, it can make it more difficult to resolve congestion. In particular, the fixed bounding box limits how far a net can 'move away' from a congested region (as it is restricted to remain inside the bounding box).

To avoid this case, the router will now dynamically expand a net's bounding box at the end of each routing iteration, provided the net uses a routing resource which is adjacent to an edge of the bounding box. This means if a net is routed to the edge of its bounding box (e.g. to avoid congestion) the next iteration will use a larger bounding box enabling it to move further out of the way. This ensures, in highly congested designs, there is no hard limit restricting how far non-critical nets can move out of the way to alleviate congestion.

Table 13 shows the impact of dynamic bounding boxes with all other routing optimizations enabled. Dynamic bounding boxes improve minimum routable channel width by 2% on the VTR benchmarks, and reduces run-time in the less congested fixed channel width ($1.3 \cdot W_{min}$) case by 2%. While the minimum channel width search time increases moderately this is not significant, as minimum channel width run-time is sensitive to small changes in routability (which may cause binary search to explore a different sequence of channel widths which may be more challenging to route). Given that both the minimum routable channel width and route time at fixed channel are improved we believe this indicates an overall routability improvement.

Congestion Mode. If the number of routing iterations is approaching the maximum allowed, VPR 8 will enter a high-effort congestion mode. This mode disables delay-driven incremental re-routing (so the router only rips-up congested connections) and also increases net bounding box sizes, giving the router more freedom to find alternative (less congested) routes.

Table 14 shows the impact of different congestion thresholds on the VTR benchmarks, with all routing optimizations enabled. Decreasing the iteration congestion threshold increases router effort in hard-to-route instances. With thresholds of 80% and 50% of maximum routing iterations respectively the minimum routable channel width decreases by 1% and 2%, while minimum channel width route time increases by 8% and 57% respectively. There is no run-time impact at relaxed channel widths since congestion is less severe.

9.3 Routing with Non-Configurable Switches

As described in Section 4.2.3, VPR 8 supports non-configurable connections between routing resources (i.e. connections which must always be used and can not be disabled), such as

[40]The default router bounding box is the net's bounding box expanded by 3 routing channels on each side.

non-tristate-able buffers (which are common in clock networks and can be useful for long wires) and electrical shorts. Using non-configurable switches to create electrical shorts allows the modeling of a broader range of wires such as more complex 'L' and 'T' shaped wires, while allowing the fundamental RR node data structure to remain a simple one dimensional wire. This ensures RR nodes can be described using compact and efficient data structures. This is very important as the RR node data structures form the largest component of VPR's peak memory footprint, and keeps the far more common case of a linear wire efficient.

The VPR router has traditionally assumed each switch, modelled as an edge in the Routing Resource (RR) graph, was configurable – meaning it could be enabled/disabled to connect/disconnect two routing resources. Supporting non-configurable switches required modifications to the router's path finding algorithm.

Algorithm 4 VPR 8 Path Finding

Require: The *heap* containing start locations, the *target* node to be found
Returns: A low-cost path to *target* from elements initially in *heap*

```
1: function FIND_PATH_FROM_HEAP(heap, target)
2:     while NOT_EMPTY(heap) do
3:         elem ← POP_SMALLEST(heap)
4:         if target = elem.node then
5:             return TRACEBACK_PATH(elem)                    ▷ Found path to target
6:         for edge ∈ OUTGOING_EDGES(elem.node) do            ▷ Explore Neighbours
7:             next_elem ← EXPAND_NODE(elem, SINK_NODE(edge), target)
8:             PUSH(heap, next_elem)
9:     return ∅                                               ▷ No path to target
```

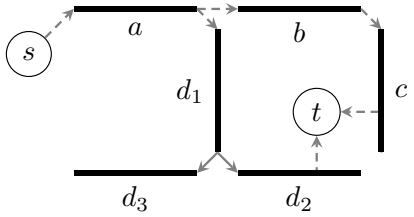
Algorithm 5 VPR 8 Node Expansion

Require: The *previous* element used to reach the current *node*, the *target* node to be found
Returns: An element to be placed on the heap with the costs of using *node* to reach *target*

```
1: function EXPAND_NODE(prev, node, target)
2:     if REACHED_BY_CONFIGURABLE_EDGE(prev, node) then
3:         nodes = NONCONFIG_CONNECTED_NODES(node)            ▷ Includes node
4:         node_congestion = SUM_CONGESTION(nodes) ▷ All non-config. connected nodes
5:     else                                  ▷ Part of previously reached non-config. node set
6:         node_congestion = 0      ▷ Already included in non-config. connected node set
7:     congestion = prev.congestion + node_congestion
8:     delay = prev.delay + DELAY(prev.node, node)            ▷ Only delay of current node
9:     γ = CRITICALITY(target)
10:    cost = (1 − γ) · (congestion + EXPECTED_CONGESTION(node, target))
11:                + γ  · (delay + EXPECTED_DELAY(node, target))
12:    return ELEMENT(node, congestion, delay, cost)
```

Algorithm 4 details VPR's algorithm for finding paths for each net connection through the RR-graph. Like a standard Djikstra/A*-based path finding algorithm VPR maintains a heap of currently explored nodes in the graph. The lowest cost node is popped off the heap (Line 3), and its neighbours are explored and added to the heap (Lines 6 to 8). This continues until either the target is found and the resulting path returned (Lines 4 and 5), or the heap is emptied (Line 2) indicating no path exists (Line 9).

There are several challenges to supporting non-configurable edges. First, the router must ensure all non-configurably connected nodes are always used together. This is achieved by



(a) Example routing resource graph with source s and target t. Dashed edges are configurable, solid edges are non-configurable.

Node	Cong.	Delay	Est. Delay to t
s	0	0	7
a	1	2	5
b	2	4	3
c	3	6	1
d_1	4	4	2
d_2	4	5	1
d_3	4	5	5

(b) Path costs from s.

Fig. 23. Non-configurable routing example. Assumes unit congestion cost and delay for wires, unit delay for configurable switches, and zero delay for non-configurable switches.

pre-processing the RR-graph to identify any sets of non-configurably connected nodes in the graph, and by updating the path traceback routine (Algorithm 4 Line 5) to add all non-configurably connected nodes to the route tree when a path is found. Second, it is crucial to cost such nodes appropriately so the router effectively optimizes their use. In particular, the router must:

- see the full congestion cost impact of using any node in a non-configurably connected set (since choosing to use at least one node from the set implies using all nodes in the set), and
- account for the delay impact of choosing particular nodes to form the routing path (since the individual nodes selected determine the delay).

Algorithm 5 shows how this is done in VPR 8. Whenever a node is explored via a configurable edge the relevant nodes are identified (Algorithm 5 Line 3)[41] and their congestion costs summed (Algorithm 5 Line 4).[42] When a node is expanded via a non-configurable edge (Algorithm 5 Line 6), by definition it is part of a non-configurably connected node set, whose congestion cost is already included (it was added to the previous congestion when the current partial path first encountered the set), so no additional incremental congestion cost is added. The remainder of the cost calculations proceed as usual, weighting the delay and congestion costs by target criticality, and including the expected congestion and delay costs to the target (Algorithm 5 Lines 9 to 11).

Figure 23 illustrates how non-configurable switches affect costing. As the router explores from the source s in Figure 23a it may expand to node d_1 which is a member of the non-configurably connected node set $\mathcal{D} = \{d_1, d_2, d_3\}$. As shown in Figure 23b, the congestion cost to reach d_1 includes the cost of using *all* members of the set, but its delay cost only reflects the delay to reach d_1. When expanding from a to d_1 the router pushes *only* d_1 onto the heap (not all of $\mathcal{D}$), but costs it to reflect the *full* congestion cost of using $\mathcal{D}$. This ensures the router does not under estimate the cost to use d_1 (since choosing to use d_1 actually requires using all of $\mathcal{D}$). If the router later pops d_1 off the heap the other members of $\mathcal{D}$ (i.e. d_2, d_3) will be expanded, accounting for their individual timing impact, but with no additional congestion cost.

It is important that each member of $\mathcal{D}$ is tracked separately in the heap, since each member may have different expected congestion/delay costs to reach the target. Tracking

[41] With pre-processing this can be done in constant time.

[42] For regular nodes this is just the individual node's congestion costs, while for a node in a non-configurably connected set this is the total congestion cost of all the nodes in the set.

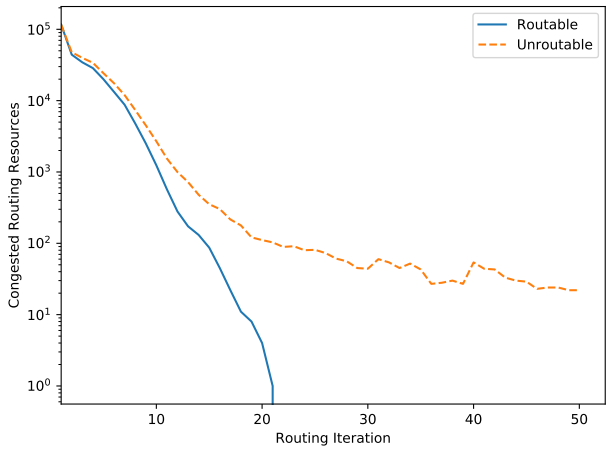


Fig. 24. Routing Congestion Trend on `LU32PEEng` at routable and unroutable channel widths

each node separately ensures the router can prioritize exploring promising nodes which move it closer to the target (e.g. expanding d_2 before d_3). This is particularly important for large non-configurably connected node sets which may fanout to a large number of nodes which may not be relevant for reaching a particular target. With this formulation the router correctly determines the path $a \rightarrow b \rightarrow c$ is better for non-timing critical connections (lower congestion cost), while $a \rightarrow d_1 \rightarrow d_2$ is better for timing-critical connections (lower delay).

9.4 Speeding-Up Minimum Channel Width Search

To find the minimum routable channel width (W_{min}), VPR invokes the router at various fixed channel widths as part of a binary search. As a result, all enhancements to the router also benefit the minimum channel width search. However VPR 8 includes several enhancements targeted particularly at reducing the run-time of minimum channel width search.

9.4.1 Routing Failure Predictor. Router run-time is highly dependent on the amount and difficulty of resolving congestion. In uncongested designs the router explores only a limited portion of the routing network and runs quickly. However, in highly congested designs the router must repeatedly explore much more of the routing network in order to find uncongested paths, which significantly increases run-time.

For designs which are unroutable this means the router will continually explore most of the routing network in a (futile) attempt to resolve congestion. To improve the run-time behaviour we designed a *routing predictor* which will abort routing early if it is not expected to complete successfully (Algorithm 2 Line 14). The routing predictor is beneficial when routing at a fixed channel width, since it reduces the time to inform the user a design is unroutable. However its largest benefit comes when an FPGA architect is attempting to find the minimum routable channel width, since the binary search run-time is dominated by routings which occur at unroutable channel widths.

As shown in Figure 24, congestion typically decreases exponentially over routing iterations.[43] The routing predictor uses this information to construct a model of congestion over time and estimate when congestion will be resolved. We use a simple linear regression model

[43]A linear trend on a log-linear plot such as Figure 24.

Table 15. Normalized Impact of Routing Predictor on VTR benchmarks ($> 10K$ primitives)

	W_{min}	Routed WL ($1.3 \cdot W_{min}$)	Crit. Path Delay ($1.3 \cdot W_{min}$)	Route Time (find W_{min})	Route Time ($1.3 \cdot W_{min}$)	VPR Memory	Total Flow Time
off	1.00	1.00	1.00	1.00	1.00	1.00	1.00
safe	1.00	1.00	1.00	0.71	1.02	1.00	0.77
aggressive	1.00	1.00	1.00	0.52	1.05	1.00	0.64

Table 16. Normalized Impact of Minimum Channel Width Hints on VTR benchmarks

	W_{min}	Routed WL ($1.3 \cdot W_{min}$)	Crit. Path Delay ($1.3 \cdot W_{min}$)	Route Time (find W_{min})	Route Time ($1.3 \cdot W_{min}$)	Total Flow Time	VPR Memory
No Hints	1.00	1.00	1.00	1.00	1.00	1.00	1.00
-30% Hint	1.02	1.00	1.00	1.10	0.97	1.05	1.00
-10% Hint	1.01	1.00	1.00	0.86	1.00	0.93	1.02
Correct Hint	1.00	1.00	1.00	0.72	0.99	0.85	1.02
+10% Hint	1.01	1.00	1.00	0.69	1.02	0.83	1.00
+30% Hint	1.02	1.00	1.00	1.09	0.98	1.04	0.99

fit using least-squares to the logarithm of the congested routing resource count.[44] At the end of each routing iteration the model is constructed using the congestion information from the most recent 50% of previous routing iterations.[45] If the estimated iteration count is greater than 3 times (`safe` setting) or 1.5 times (`aggressive` setting) the maximum number of allowed routing iterations, the routing is aborted.

Table 15 shows the impact of the routing predictor on routing QoR and run-time metrics on the large circuit subset of the VTR benchmarks, with all other routing optimizations enabled. For the safe and aggressive predictor settings there are no changes in quality metrics, but minimum channel width route time is reduced by 29% and 48% respectively.

9.4.2 Minimum Routable Channel Width Hints. Often when performing minimum routable channel width experiments the architect has some insight into what the expected channel widths will be.[46]

VPR 8's minimum channel width search can now take such estimates as optional 'hints' to the minimum channel width binary search. These hints are used as the starting point of the binary search, and if accurate allow a large part of the search space to be quickly eliminated. Additionally, if the hint value is found to be routable a smaller initial step (10% channel width reduction, instead of the default 50%) is taken in hopes of quickly identifying an unroutable channel width and pruning the number of such channel widths explored.

Table 16 shows the impact of minimum channel width hints with all other routing optimizations enabled. Providing accurate hints (`Correct Hint`) results in the same minimum channel width as not providing a hint (`No Hint`), but reduces minimum channel width search time by 28%. It should be noted that the accuracy of the hints does not significantly affect the resulting minimum channel width found by the binary search. Less accurate hints can still reduce minimum channel width search time (`+10% Hint`, `-10% Hint`). Inaccurate hints (`+30% Hint`, `-30% Hint`) only moderately increase search time since the number of additional channel widths explored is bounded.

[44] Using the logarithm allows a linear model to fit an exponential trend.
[45] Using the most recent 50% of routing iterations means only recent history is considered, but the length of history increases as routing proceeds. This helps minimize noise caused by the typically small number of congested routing resources late in the routing process.
[46] For instance they may have an estimate based on prior experience with the same or similar architectures.

Table 17. VPR Router Comparisons on Titan v1.1.0 Benchmarks

	# Routable Benchmarks	Routed WL	Routed CPD	Route Time
VPR8 Timing-Driven (Timing + WL)	21	1.00×	1.00×	1.00×
VPR8 Timing-Driven (WL only)	21	0.97×	1.47×	0.70×
VPR8 Breadth-First (WL only)	14	1.10×	1.66×	394.54×
VPR7+ Timing-Driven (Timing + WL)	20	1.18×	1.16×	6.16×
VPR7+ Breadth-First (WL only)	14	1.08×	1.56×	304.45×

All routers used identical packings/placements/RR graphs produced by VPR 8.
QoR is the geomean of the mutually routable subset of 14 benchmarks.

9.5 Router Algorithm Comparison

To evaluate the effect of the router enhancements, Table 17 compares the various routing algorithms in VPR 7+ and VPR 8 under identical conditions (same netlist, packing, placement, and RR graphs produced by VPR 8) on the Titan benchmarks [85]. VPR contains two different routing algorithms the primary *timing-driven* router (whose enhancements in VPR 8 are described above) which can be run to either optimize wirelength and timing simultaneously (Timing + WL) or wirelength only (WL only), and the *breadth-first* router which optimizes wirelength only and is included for historical reasons [15].

First considering the VPR 8 and VPR 7+ timing-driven routers (Timing + WL), we observe the VPR 8 router is substantially more run-time efficient (6.2× faster) while also improving QoR (18% wirelength and 16% critical path delay reductions). It also successfully routes an additional benchmark compared to VPR 7+. Second, the simpler breadth-first routers in VPR 7+ and VPR 8 both run more than two orders of magnitude slower ($> 300\times$) than the VPR 8 timing-driven router. They also only successfully route 14 (vs. 21) benchmarks, while using 8 to 10% more wiring and produce 56 to 66% longer critical paths. Third, the VPR 8 timing-driven router is very efficient at trading-off wirelength and timing. When run with a wirelength only objective (WL only) which ignores timing, it only further reduces wirelength by 3% and run-time by 30%. This shows that when run with a combined timing and wirelength objective, the timing-driven router is able to effectively prioritize the routing of timing critical signals without significantly degrading the routing of other connections. As a result, there is little reason to running with only a wirelength objective.

These results show the VPR 8 timing-driven router is superior in both run-time and QoR to the breadth-first routers and the VPR 7 timing-driven router. Therefore other works building on and/or comparing to the VPR should ensure they compare to the VPR 8 timing-driven router. Choosing to compare to the non-default breadth-first router will result in misleading conclusions. For instance, parallel routing algorithms typically report speed-ups of 2 to 8× [105], however not all authors use the same serial baseline. While some works have used the VPR 7 timing-driven router as a baseline, others used the VPR 7 breadth-first router which is orders of magnitude slower. This means not all published parallel routing speed-ups are significant. Going forward, parallel routing researchers should ensure they use a state-of-the-art serial routing algorithm as their baseline.

10 TIMING ANALYSIS

VTR 8 uses Tatum [81] as its Static Timing Analysis (STA) engine, which was designed to be more complete, flexible and higher performance than VPR 7's classic STA engine. Tatum provides various new timing analysis features such as hold (min-delay) timing analysis, support for multi-clock netlist primitives (e.g. dual port RAMs with different clocks), additional timing constraints and exceptions, and support for paths completely contained

Table 18. Normalized Impact of Setup & Hold STA Traversals While Routing Titan Benchmarks

	STA Traversal Time
Separate	1.00
Combined	0.70

Table 19. Comparison of Classic & Tatum STA (Setup Only) on Titan Benchmarks

	Pack Time	Place Time	Route Time	Total Time	STA Time
Classic	1.00	1.00	1.00	1.00	1.00
Tatum serial	1.00	0.88	0.95	0.94	0.47
Tatum parallel	0.99	0.80	0.91	0.90	0.07

within primitives (such as registered block RAM read paths). As a result, VTR 8 also supports more advanced timing reports, including detailed multi-path timing reports which help users understand and evaluate the speed-limiting paths in their design. Furthermore, Tatum produces the same analysis results as VPR 7 (to within floating point round-off) for setup analysis, allowing critical path delays to be fairly compared between VPR 7 and VPR 8 when identical delay models are used.

As STA is called hundreds of times to evaluate the timing characteristics of the design implementation and drive further optimization, it is key for STA to be fast. Tatum includes several enhancements which improve its performance over VPR 7's classic STA engine.

- Tatum performs a single traversal of the timing graph, simultaneously analyzing all clock domains. This is more efficient than VPR 7's classic STA engine, which traversed the timing graph for each pair of clock domains.
- If both setup and hold timing analyses are required (e.g. during combined long and short path timing optimization), Tatum can perform a combined setup and hold analysis during the same traversal of the timing graph.
- Tatum supports parallel execution across multiple cores to further speed-up STA.

10.1 Combined Setup & Hold Analysis

Table 18 quantifies the effect of combining setup and hold analyses into a single traversal (`Combined`) compared to performing two independent traversals (`Separate`) while routing the Titan benchmarks. Performing a single set of combined traversals reduces the total STA traversal time by 30% due to improved cache locality. With the combined traversals the timing graph does not need to be reloaded into the cache for both setup and hold analyses.

10.2 Comparison with VPR Classic STA

Table 19 compares the run-time of using VPR's classic timing analyzer and Tatum when implementing the Titan benchmarks. Overall STA time in VPR is reduced by 2.1× serially and 14.3× in parallel with 24 cores.[47] As a result overall run-time is reduced by 6% (`Tatum serial`) to 10% (`Tatum parallel`), with the largest run-time reductions occurring in placement (12-20%) and routing (5-9%). However it is worth noting that for some designs STA is a larger proportion of overall run-time, and the run-time reductions are larger.[48]

[47] Compared to VPR's classic STA engine, Tatum performs best on large multi-clock designs, with circuits like `mes_noc` and `gsm_switch` running 7.5-6.2× faster serially, and 59.6-48.0× faster with 24 cores.
[48] For example, `stereo_vision`'s total run-time is reduced by 21% using Tatum with parallelism.

11 SOFTWARE ENGINEERING

VTR is a large complex software project, and as a result requires significant software engineering efforts. These efforts are multifaceted and include:

- Optimizing the implementations of key algorithms and data structures,
- Improving robustness and stability,
- Providing support and documentation for users, and
- Facilitating collaboration between developers.

Some of these items have a significant impact on the technical merits of the project such as Quality of Results (QoR) and run-time, while others affect VTR's usefulness to end-users, and the project's longer-term health and sustainability. This section highlights a subset of this work.

11.1 ODIN II

11.1.1 HDL Elaboration. Odin II (developed in C/C++) has been extended to support sequential circuits with more complex control signals (e.g. negative edge sensitive and asynchronous). This expands the capacity of VTR 8.0 to benchmark a wider variety of circuits. Verilog language coverage has been extended, a wide variety of bugs have been fixed, and optimizations like constant folding have been improved.

The addition operator in Verilog can be synthesized in a number of functionally equivalent ways. In VTR 8, Odin II supports producing a variety of adder implementations with different power, area and delay trade-offs [69].

Odin II also has improved support for identifying and removing unused/redundant logic as described in [83]. This is particularly import when the netlist contains blackbox primitives, as logic optimization tools like ABC can not sweep logic related to such primitives (since ABC has no visibility into the functionality of the blackboxes).

11.1.2 Simulator. Odin II includes a simulation tool that allows vector-based verification of both Verilog and BLIF circuits. The user can provide a set of input and output vector files which describe successive input signals that drive the circuit and the expected circuit outputs.

In VTR 8 we extended the simulator to support multi-clock circuits and additional sequential elements (asynchronous, falling edge and rising edge). The simulator can also generate more compact test-vectors for a circuit, by using the current test coverage to guide future vector generation [90].

11.2 VPR

VPR is developed in C/C++, and has a long development history, being continually extended and enhanced over time. As a result it is important to improve code quality, which makes it easier for both developers and end-users to make future enhancements and modifications to VPR. Additionally, for a high performance CAD tool solving large-scale optimization problems, the implementation details of various data structures and algorithms have a significant impact on run-time and memory footprint. These enhancements are beneficial to the research community as it makes it easier for researchers to develop and evaluate new CAD algorithms within VPR, or customize VPR to explore new architectural features.

11.2.1 VPR Netlist. One of the significant software engineering efforts in VPR 8 was refactoring the netlist data structures which store the technology mapped (or 'atom') netlist, and the post-packing 'clustered' netlist.

Table 20. Normalized Impact of Atom Netlist Data Structure Refactoring on VTR Benchmarks

	Pack Time	Place Time	Route Time (find W_{min})	Route Time ($1.3 \cdot W_{min}$)	Total VPR Time
Baseline	1.00	1.00	1.00	1.00	1.00
Refactored	0.63	1.00	0.92	0.93	0.86

These data structures have been significantly re-written in order to produce a well-defined common API for accessing and modifying the netlist data. During this process the underlying data layout and representation was modified to reduce indirect memory accesses (pointer-chasing), and improve cache utilization. In particular, the data layout was changed from an Array-of-Structs (AoS) to Struct-of-Arrays (SoA) representation, which is more cache friendly when only a few data fields are accessed at a time.

Table 20 shows the atom netlist refactoring reduced pack-time, route-time and total VPR run-time by 37%, 7-8% and 14% respectively.

11.2.2 Contexts. VPR has historically made use of numerous global variables to represent the device model and design implementation state, which made it challenging to reason about and modify the code base.

To address this we have restructured and grouped together the various data structures representing the device model and implementation state into *contexts*. The device context contains all data structures used to represent the targeted FPGA device, including block type definitions, device grid and RR-graph. There are also contexts representing different aspects of the implementation state such as the clustering, placement or routing. Each part of the code base (e.g. the router) now explicitly asks for the different contexts they require, and specifies whether read-only or read-write access is required. This helps to minimize coupling between different parts of the code base, while making the dependencies more explicit.

11.2.3 Redundant Work. After performing a move during placement, VPR iterates through the affected nets to update wirelength and timing information. In VPR 7, the affected nets would be iterated through multiple times. We have improved this in VPR 8, by ensuring that all updates required occur during a single iteration through the affected nets. Similarly to performing a single set of timing graph traversals (Section 10.1), this improves cache locality by not repeatedly reloading the netlist and placement information into the cache.

11.2.4 Memory Usage Optimizations. It was previously noted that VPR 7's peak memory usage was substantially higher than commercial tools like Quartus [85], which required very high memory computers to implement the largest FPGA designs. As a result we have worked to reduce peak memory usage.

The largest data-structure in VPR is typically the RR-Graph, which can contain millions of nodes and 10s of millions of edges. This data structure is carefully packed to minimize the size of each node and edge (improving cache utilization), and common data is factored out in a flyweight pattern [41].

Table 21 shows the impact of the most recent set of optimizations on the Titan `neuron` benchmark. VPR's peak memory usage typically occurs during RR-Graph construction, as both the RR-graph and auxiliary RR-graph-sized data structures (used to build the RR-graph) are both allocated. During RR-graph construction, exactly sizing allocations and allocating contiguous regions of memory helped to significantly reduce memory usage caused by fragmentation, as did flyweighting additional non-unique data. Moving to a sparse representation of the intra-block routing within each cluster also reduced memory usage,

Table 21. Cumulative Impact of VPR Peak Memory Optimizations on `neuron`

	Code Area	Peak Memory (MiB)	Ratio
Baseline		9,030	1.00
Exact RR Graph alloc.	RR Construction	5,874	0.65
Contiguous SB pattern	RR Construction	5,423	0.60
Flyweight segment details	RR Construction	4,573	0.51
Sparse Intra-block routing	Packer	3,866	0.43
Contiguous RR Edge & RR Node fanin	RR Construction	3,431	0.38

Table 22. Normalized Impact of Advanced Compiler Optimizations on VPR 8.0 Run-Time (Titan Benchmarks)

	Pack Time	Place Time	Route Time	Misc. Time	Total Time	STA Time
Standard	1.00	1.00	1.00	1.00	1.00	1.00
IPO	1.00	0.77	0.74	0.93	0.81	0.76
IPO + PGO	0.89	0.70	0.62	0.92	0.74	0.68

STA Time is included in Pack/Place/Route/Misc. and Total Times

since much of the routing is unused in any particular cluster. These optimizations reduced peak memory use by 2.6×. As discussed in Section 12.2 VPR's peak memory usage is now comparable to Quartus on the Titan benchmarks.

11.3 Compiler Settings

Modern C++ compilers support a variety of advanced optimizations such as Inter-Procedural/ Link-Time Optimization (IPO/LTO) and Profile-Guided Optimization (PGO). Table 22 shows the impact of building VPR with these optimizations enabled using GCC 8. IPO offers significant improvements reducing place, route and STA time by 26-23% and total time by 19%. PGO (using a profile generated from a run of `neuron`) improves the total run-time reduction further to 26%. Given that the run-time reductions from these compilation options are significant it is important for future work (e.g. comparing to VTR) to ensure they use consistent compilation settings, so as to not miss-attribute run-time improvements to algorithmic changes when they are in fact due to compiler settings.

11.4 Regression Testing

VTR has a large code base consisting of several different tools, each of which have a large number of features. To ensure robustness it is important to have tests which catch when existing features are broken by code changes. Additionally, it is vital to track VTR's Quality of Results (QoR) to detect any quality or run-time regressions, as it is otherwise very easy to inadvertently degrade VTR's optimization algorithms or data structures.

The full regression test suite requires 168 hours to run serially, and covers over 370 architecture, benchmark and tool option combinations. Since each combination is independent the run-time can be reduced to less than 40 hours with moderate parallelism (3 or 4 cores).

11.4.1 Benchmark & Architecture Coverage. VTR 8 regression tests now cover a wider variety of architectures including: classic (soft-logic only) architectures, modern FPGA architectures like the VTR Flagship and Titan architectures, architectures with custom hard blocks, and architectures using a variety of different routing technologies (classic bi-directional, uni-directional, complex custom switch patterns). We now also test a wider variety of benchmark sets including the classic MCNC20 [124], VTR benchmarks, Titan 'other' and Titan23. In particular including the Titan23 ensures VTR's QoR and scalability are regularly evaluated on large complex benchmarks.

Table 23. VTR Code Coverage

	Norm. Lines		Test Coverage	
	VTR 7	VTR 8	VTR 7	VTR 8
odin	1.00	1.07	0.70	0.69
vpr	1.00	1.40	0.75	0.86
vpr_base	1.00	1.26	0.58	0.86
vpr_pack	1.00	1.25	0.85	0.86
vpr_place	1.00	1.17	0.92	0.94
vpr_route	1.00	2.40	0.79	0.84
vpr_timing	1.00	0.75	0.68	0.87
vpr_power	1.00	1.19	0.88	0.87
vpr_util	1.00	1.96	0.77	0.77
libarchfpga	1.00	1.20	0.70	0.80
libvtrutil				0.80
libtatum				0.82

11.4.2 Feature Coverage. In addition to covering additional architectures and benchmarks, VTR 8 also includes more extensive tests which verify the functionality of specific features. While most VTR tests are integration tests that test the overall functionality of the flow or specific tools, we have also begun including unit tests to verify the functionality of smaller parts of the code-base.

While not a perfect metric, code-coverage quantifies how well the test suite covers the various parts of the code-base. Table 23 compares VTR 7 & 8's code line count and code-coverage.[49] Compared to VTR 7, the code base for Odin II has grown by 7% while VPR has grown by 40%. The large growth in VPR is due to the additional architecture modelling features, improved code structure, better encapsulated data structures, and enhancements to the various optimization algorithms (particularly the router). Despite this growth VPR's overall code coverage has increased to 86% with the largest coverage improvements occurring in the router and base functionality.

11.4.3 Formal Verification. In addition to integration and unit tests which measure the functionality and QoR of VTR, in VTR 8 we have also added correctness tests. These leverage VPR's improved netlist writer and ABC's formal verification capabilities (Section 3.1.3) to formally prove the resulting implementation is equivalent to the original netlist. These tests ensure the various tools correctly implement the design and do not change the design's functionality.

11.5 Error Messages & File Parsers

A common issue encountered by users of previous versions of VTR has been the limited number of (or unhelpful) error messages. We have made significant efforts in VTR 8 to provide more detailed, easier to interpret and actionable error messages.

A large part of this effort has involved improving the robustness of VPR's file format parsers, particularly those dealing with human-editable formats such as the FPGA Architecture file, SDC constraints, and BLIF netlist. To facilitate this several of the file parsers (SDC, BLIF) have been re-written using a formal format specification and parse generators [2, 3] which leads to helpful error messages when invalid syntax is provided. The FPGA architecture file has also been updated to use a robust XML parsing library [5] which removes artificial syntax restrictions, while the logic which interprets the XML file has been improved to emit more user-friendly error messages when invalid or unexpected values are encountered.

[49] Results were collected with the `gcov` utility [4].

Table 24. Summary VTR Flagship Architecture Characteristics

Metric	Expression	Value
Process Technology		40nm
LUT Size	K	6 (fracturable)
BLEs per Logic Block	N	10
Adder Bits per Logic Block		20
Memory Block Size		32 Kb
DSP Block Multiplier Size		36x36 (fracturable)
Switchblock Type		Wilton
Wiretype Length		4

Table 25. VTR 7 & VTR 8 Comparison Summary on the VTR Benchmarks

	Netlist Primitives	ABC Depth	CLBs	W_{min}	Routed WL ($1.3 \cdot W_{min}$)	Routed CPD ($1.3 \cdot W_{min}$)	Odin Time	ABC Time	Pack Time	Place Time	Route Time (Find W_{min})	Route Time ($1.3 \cdot W_{min}$)	Total Time	Peak Memory
VTR 7: ODIN7/ABC'12/VPR7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ODIN8/ABC'12/VPR7	0.86	0.99	0.88	1.00	0.88	0.98	1.29	0.73	0.87	0.97	1.15	0.96	1.10	0.90
ODIN8/ABC'12/VPR8	0.86	0.99	1.13	0.83	0.74	0.97	1.68	0.72	0.55	0.63	0.10	0.33	0.15	0.49
VTR 8: ODIN8/ABC'18/VPR8	0.74	0.89	0.74	0.85	0.59	0.88	1.53	5.15	0.58	0.56	0.08	0.30	0.19	0.30

Values are normalized geomeans over mutually completed benchmarks

11.6 Documentation

We have also worked to improve VTR's documentation, which is key for helping users understand and use VTR. VTR's documentation is now centralized and automatically generated from the VTR source tree, ensuring it is easy to update and keep synchronized with the source code. The documentation is now searchable and available online at docs.verilogtorouting.org, and includes:

- Tutorials for common VTR use cases,
- Design flow documentation,
- Tool specific documentation,
- Common file-format specifications, and
- Developer guidelines.

The resulting VTR 8 user manual runs over 250 pages.

12 EXPERIMENTAL EVALUATION

We now evaluate the overall QoR and run-time of VTR 8. Unless otherwise noted all results were collected on an identical system with two Intel Xeon gold 6146 CPUs (24 cores) and 768GB of RAM. VTR was compiled with GCC 8 with full optimization (`-O3`) with IPO and PGO using a profile generated from single runs of the `stereovision1` and `neuron` benchmarks.

12.1 VTR Benchmarks

We now evaluate VTR 8's QoR in an architecture exploration context. These results were collected on the VTR benchmarks while targeting VTR's flagship `k6_frac_N10_frac_chain_mem32K_40nm` architecture, whose characteristics are listed in Table 24. Each benchmark is run through the standard VTR flow (Figure 1), starting from a behavioural Verilog netlist which is synthesized by ODIN II and optimized by ABC. VPR then performs packing and placement, followed by a binary search over channel width (W) to find the minimum routable channel width (W_{min}). Wirelength and critical path delay are then measured after routing at a relaxed channel width set to $1.3 \cdot W_{min}$.

Table 25 summarizes VTR 8's achieved QoR relative to VTR 7 for benchmarks with $> 10K$ netlist primitives (to avoid small benchmarks skewing the results). We include results for both the default VTR 8 (ODIN 8, ABC'18, VPR 8), VTR 7 (ODIN 7, ABC'12, VPR 7) and hybrid flows mixing tool versions, which allow us to isolate the impact of improvements

to ODIN, ABC, and VPR separately. We first evaluate the impact of enhancements to ODIN (ODIN 8/ABC'12/VPR7), which show ODIN produces fewer netlist primitives due to improved sweeping of unused logic and hard blocks [83]. We next compare the impact of enhancements to VPR (ODIN8/ABC'12/VPR8), which shows significant improvements in minimum routable channel width, wirelength and drastic improvements in pack/place/route run-times and peak memory usage, with the overall flow completing significantly faster. Finally we can compare the full VTR 8 (ODIN8/ABC'18/VPR8), showing overall a 15% reduction in W_{min}, a 41% reduction in routed wirelength and a 12% reduction in critical path delay. Additionally, the run-times of the physical optimization stages are substantially reduced (1.7× packing, 1.8× placement, 12.5× to find W_{min} and 3.3× routing at $1.3 \cdot W_{min}$), reducing total flow run-time by 5.2×. Peak memory usage is also reduced by 3.3×. Notably, while the newer ABC performs better optimizations its run-time is also substantially increased. These results show that improvements to all stages of the CAD flow contribute to the overall QoR and run-time improvements.

Table 27 shows VTR 8's detailed QoR on the VTR benchmarks and the normalized results compared to VTR 7. We again focus on the results for benchmarks with $> 10K$ netlist primitives.

The improved logic sweeping and optimization in VTR 8 reduces the number of netlist primitives by 26% and logic depth by 11%. As a result, despite decreasing packing density (Section 7) VTR 8 uses 26% fewer CLBs than VTR 7. VTR 8 produces significantly better implementation quality than VTR 7, improving minimum routable channel width by 15%, reducing wirelength by 41%, and reducing critical path delay by 12%.

These quality improvements are achieved while cutting the overall VTR flow run-time by 5.2×, and peak memory consumption by 3.3×. The run-time gains come primarily from reductions in minimum channel width search time (12.5× faster), and place time (1.8× faster). Pack and relaxed channel width route times were also reduced by 1.7× and 3.3× respectively.

ODIN II's run-time increased by 1.5× but remains a very small fraction (0.6%) of overall run-time. While the new version of ABC performs better optimizations, it runs substantially slower (5.2×) and accounts for 39% of overall run-time in the architecture exploration flow (find W_{min}), which is comparable to the time spent on the physical implementation (packing, placement and routing). ABC run-time is an even larger fraction (62%) of run-time in the design implementation ($W = 1.3 \cdot W_{min}$) flow. As a result, despite significant run-time improvements during the physical implementation, VTR 8's run-time is comparable to VTR 7's when targeting a fixed channel width.

12.2 Titan23 Benchmarks

To compare VTR and Intel's commercial Quartus [31] tool we used the Titan Flow (Figure 1), where designs are synthesized and technology mapped using Quartus. The physical implementation (packing, placement, routing) can then be performed using either Quartus or VPR.

Unless otherwise noted, VPR was run with its default options, except the maximum number of routing iterations was increased to 400, and the router used the new map lookahead (Section 9.2.1) in VPR 8. For all benchmarks a 48 hour wall-clock run-time limit was imposed. Any benchmarks exceeding the limit were classified as failures. Critical path delay is reported as the geometric mean over all clock domains in a circuit, excluding any virtual I/O clocks.

12.2.1 Comparison with VPR 7. To compare VPR 7 and 8 on the Titan benchmarks, we used VPR 8 and VPR 7+.[50] Both tools use the same Titan v1.1.0 benchmarks synthesized with Quartus 12.1, and target the same legacy Stratix IV architecture model from [85].

Table 28 compares VPR 8 and VPR 7+ on the Titan v1.1.0 benchmarks. VPR 8 completes 5 more benchmarks than VPR 7+, showing that it is more robust and better able to handle large complex designs. VPR 8 packs less densely, using 6% more LABs and 13% more DSPs than VPR 7+ (Section 7.2.2). VPR 8 reduces routed wirelength by 34% and critical path delay by 11% on the common benchmarks which completed in both tools.

From an overall run-time perspective VPR 8 runs substantially faster (5.3×) and uses much less memory (5.0×) than VPR 7+. The largest run-time improvement came from routing which completes 33.3× faster in VPR 8, highlighting the utility of the router enhancements detailed in Section 9. Pack time is also reduced by 1.5× and place time by 1.1×.[51]

12.2.2 Comparison with Quartus 18.0. To compare VPR and Quartus on the Titan benchmarks, we used VPR 8 and Quartus 18.0. Both VPR and Quartus used the more recent Titan v1.3.0 benchmarks synthesized and technology mapped with Quartus 18.0. Quartus 18.0 targeted Stratix IV, while VPR 8 targeted the Stratix IV architecture model from Section 5 (which improves upon the model used in [85]). However the precise details of the Stratix IV routing architecture are not publicly available, so it is not possible to perform a perfect comparison.[52] Our Stratix IV model uses a timing model calibrated to Quartus STA's reported component delays, and models the key characteristics of both block and routing architectures and was validated in [85]. We believe this allows us to perform a reasonable comparison between VPR and Quartus.

Quartus is configured to use 'auto' device selection which selects the smallest device with sufficient resources to implement the design. Similarly, VPR will automatically construct the smallest device with sufficient resources to implement the design. Since several of the Titan designs are larger than the largest commercially available Stratix IV device Quartus refuses to fit them, while VPR can build a sufficiently large device.

Quartus was run in `STANDARD_FIT` mode to ensure full optimization effort. To ensure run-times remain comparable, both Quartus and VPR were run using a single thread. VPR was run at its default effort level and at an increased High Effort (HE) level (`inner_num` = 2, `astar_fac` = 1) which resulted in run-times comparable to Quartus.

Both Quartus and VPR were given equivalent timing constraints. Paths between clock domains were cut, except for those to/from external I/Os which are constrained on a virtual I/O clock. Each clock domain is given an aggressive 1ns clock period target. Extremely small clock domains (those fanning out to $< 0.1\%$ of netlist primitives) were ignored to avoid skewing the average critical path delay and focus on primary system clocks.

Table 26 summarizes the QoR comparison between various versions of VPR normalized to Quartus. Compared to VPR 7+, VPR 8 completes significantly more circuits (23 vs. 14). When VPR 8 is run with its default parameters it requires 38% more wire and produces circuits which are 32% slower than Quartus 18.0, but runs nearly 2× faster and uses less memory. Increasing VPR 8's effort level so run-time is comparable to Quartus 18.0, and including all 20 benchmarks which mutually complete (`VPR 8 HE / Quartus 18.0`) the

[50] VPR 7+ is the enhanced version of VPR 7 used in [85] which is compatible with the Titan v1.1.0 benchmarks.

[51] Miscellaneous time increases in VPR 8 since the routing graph needs to be profiled to create the map lookahead, but remains a small fraction overall.

[52] We expect VPR's automatically generated RR graph, while matching Stratix IV's major characteristics, is of lower quality than the extensively hand-optimized pattern used in Stratix IV.

Table 26. VPR Quartus QoR Summary on Titan Benchmarks

	VPR # Benchmarks Completed	Quartus # Benchmarks Completed	LABs	DSPs	M9Ks	M144Ks	Routed WL	Routed CPD	Pack Time	Place Time	Route Time	Total Time	Peak Memory
VPR 7+ / Quartus 12.0 *	14	20	0.82	1.13	1.13	1.69	2.02	1.53	2.36	0.87	8.23	2.82	6.21
VPR 8 / Quartus 18.0 *	23	20	0.85	0.96	1.33	0.00	1.38	1.32	1.39	0.46	0.30	0.51	0.96
VPR 8 HE / Quartus 18.0 *	23	20	0.85	0.96	1.33	0.00	1.31	1.26	1.39	1.04	0.35	0.87	0.97
VPR 8 HE / Quartus 18.0	23	20	0.95	0.97	1.30	0.00	1.26	1.20	1.18	1.00	0.34	0.83	0.96

* QoR from common subset of 13 completed benchmarks across all tool pairs
VPR 7+/Quartus 12.0 results from [85].

wirelength and circuit speed gaps narrow to 26% wirelength and 20% critical path delay.[53] VPR 8 substantially narrows the quality gap, with the wirelength and critical path delay gaps reduced by 1.5× and 1.2× respectively compared to VPR 7+. Notably, including the larger benchmarks which VPR 8 can now complete further narrows the gap with Quartus, indicating the scalability of VPR's optimizations. VPR 8's run-time is also substantially improved (5.5× to 3.2× faster overall compared to VPR 7+) with route-time seeing the largest improvement (27× to 24× faster), while peak memory use is reduced by 6.5×.

Table 29 compares VPR 8 (in high effort mode) and Quartus 18.0 in detail. VPR 8 completes all 23 of 23 feasible Titan benchmarks, and Quartus completes 20 of 21 feasible benchmarks. This shows that VPR and Quartus' benchmark completion rates are similar.

In terms of resource utilization VPR 8 uses 5% fewer LABs and 3% fewer DSPs than Quartus. However VPR 8 uses more M9Ks and fewer M144Ks since it uses the larger and rarer M144Ks only when RAMs would otherwise not fit in M9Ks. Due to resource utilization differences VPR 8's auto-sized devices can either smaller or larger than the devices used by Quartus, but on average they are 15% larger. As expected, those designs where VPR uses more resource of relatively sparse types (e.g. M9K, DSPs) than Quartus tend to produce the largest differences in device sizes.

From a run-time perspective, VPR 8 HE runs 1.2× faster than Quartus and has a 4% smaller memory footprint. VPR 8's packer runs 18% slower than Quartus', but is much more general purpose. For these large benchmarks, placement dominates run-time (78% of total time), and VPR 8's placer run-time is comparable to Quartus. The VPR 8 router is more efficient and runs 2.9× faster than Quartus even in this high effort mode. Compared to VPR 8, Quartus spends approximately 3.3× more run-time on STA.

VPR 8's QoR and run-time improvements in VPR 8 result from two key factors:

(1) CAD algorithm enhancements, which improve the quality and robustness of VPR's optimizations, and
(2) Improved architecture modelling capabilities (Section 4), which allow VPR to target a more realistic model of Stratix IV (Section 5), and in particular a more accurate capture of Stratix IV's routing architecture.[54]

This shows the benefit of simultaneously improving both architecture modelling capabilities and CAD algorithms together.

13 CONCLUSION

We have presented version 8.0 of the open-source Verilog To Routing project. VTR 8 includes substantial enhancements to VTR's architecture modelling capabilities allowing VTR to model a wider range of modern FPGA characteristics including: general device grids, highly customizable routing architectures, and improved area and delay estimates. These features

[53] For multi-clock circuits geometric mean and worst critical path delay results were similar.
[54] However it should be noted the detailed routing pattern used by VPR is still automatically generated, and likely remains of lower quality than the heavily hand optimized routing pattern used in Stratix IV.

can be used to both facilitate new research (e.g. by allowing more control over characteristics such as the detailed routing pattern), and to better model and target modern commercial FPGAs with suitable bitstream generators [6].

In addition to these features we have also significantly enhanced the CAD algorithms and tools used to implement and optimize designs in VTR. These enhancements have significantly improved implementation quality of the VTR benchmarks (15% minimum channel width, 41% wirelength, and 12% critical path delay reductions) while also reducing tool run-time and memory usage by 5.2× and 3.3× respectively.

On the larger industrial-scale Titan benchmarks, VPR 8 now completes a similar proportion of mutually feasible designs as a commercial tool like Quartus. We have also reduced the quality gap between VPR and the commercial Quartus tool to 26% wirelength and 20% critical path delay (reductions of 1.5× and 1.2× compared to previous results), while simultaneously running 17% faster and using similar amounts of memory.

These results show that through careful code architecture and algorithm design it is possible for general re-targetable CAD tools like VPR to be both run-time and memory footprint efficient, while producing circuit implementations which are of reasonable quality compared to those produced by highly tuned architecture-specific industrial tools. Reducing the implementation quality gap also helps ensure the architectural conclusions produced using these tools are valid and not skewed due to poor optimization.

14 FUTURE WORK

In the future we plan to continue extending VTR's modelling capabilities to enable the exploration of an even broader range of FPGA architectures and to enable full modelling of advanced features found in commercial FPGAs.

We will also continue to focus on improving the quality of VTR's optimization algorithms. We expect that a large portion of the remaining quality gaps result from clustering quality. In particular, while VPR 8 produces clusters which are more natural than VPR 7, related logic can still sometimes be scattered between different clusters, which end-up placed far apart. The result is that such signals end-up crossing large parts of the chip and increasing wirelength. Inspection of the resulting critical paths shows a similar trend, with timing critical connections sometimes spanning large portions of the chip. We believe it is important to consider cross-boundary optimizations between the traditional pack/place/route stages, such as being able to take-apart clusters during placement [25].[55] It would also be interesting to continue exploring the application of Machine Learning [82, 127] techniques to various stages of the CAD flow. From a run-time perspective further investigation into hierarchical and parallel compilation techniques also seem promising.

We also plan to continue extending VPR to be an end-to-end CAD flow capable of both targeting existing FPGA devices and facilitating the creation of physical devices based on VTR architectures. To that end, it would be useful to extend the RR graph file (Section 3.1.2) to become a description of the entire FPGA device (including device grid and block architectures). Such a 'device file' could be generated from VTR's higher-level architecture file description, or through other means to model new architectures or architectural features not currently supported by VTR's higher-level descriptions. This would allow VTR's design implementation and optimization algorithms to be better decoupled from the high-level architecture description, and be more easily re-targeted to other architectures.

[55] Previous measurements of Quartus [85] indicate taking apart clusters allows Quartus to reduce wirelength and critical path delay by 9% and 10% respectively.

While we believe that VTR's improved optimization quality allows it to make accurate CAD and architecture conclusions, it would be interesting to verify its fidelity by, for instance comparing the impact of different optimizations in both VTR and commercial CAD systems.

It would additionally be interesting to compare the quality of routing patterns generated by VTR when targeting our model of Stratix IV compare to the heavily optimized actual routing patterns used in commercial devices such as Stratix IV. It is also future work to consider how VTR could be extended to support multi-context FPGAs, and partial reconfiguration.

ACKNOWLEDGEMENTS

This work was supported by the NSERC/Intel Industrial Research Chair in Programmable Silicon, Huawei, Lattice Semiconductor, the Semiconductor Research Corporation, the Canadian Foundation for Innovation, the New Brunswick Innovation Foundation, an NSERC CGS-D scholarship, and an Ontario Graduate Scholarship. The authors would also like to thank Jeff Goeders, Sadegh Yazdenshenas, Maria Patrou, Alex Demmings, Mustafa Abbas, and all those who have contributed to the VTR project by submitting patches/pull-requests and bug reports including: Tim Ansell, Keith Rothman, Alessandro Comodi, and David Shah. Compute resources were provided in part by Compute Canada [19].

REFERENCES

[1] 2019. Arachne-pnr. https://github.com/YosysHQ/arachne-/pnr
[2] 2019. Bison. https://www.gnu.org/software/bison/
[3] 2019. Flex: The Fast Lexical Analyzer. https://www.gnu.org/software/flex/
[4] 2019. gcov: A Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[5] 2019. Pugixml: Light-weight, simple and fast XML parser for C++. https://pugixml.org/
[6] 2019. SymbiFlow Project. https://symbiflow.github.io/
[7] M. S. Abdelfattah and V. Betz. 2012. Design tradeoffs for hard and soft FPGA-based Networks-on-Chip. In *Int. Conf. on Field-Programmable Technology (FPT)*. 95–103.
[8] Ziad Abuowaimer, Dani Maarouf, Timothy Martin, Jeremy Foxcroft, Gary Gréwal, et al. 2018. GPlace3.0: Routability-Driven Analytic Placer for UltraScale FPGA Architectures. *ACM Trans. Des. Autom. Electron. Syst.* 23, 5, Article 66 (Oct. 2018), 33 pages.
[9] Achronix Semiconductor 2019. *Speedster7t FPGAs*. Achronix Semiconductor. PB003 v1.0.
[10] I. Ahmadpour, B. Khaleghi, and H. Asadi. 2015. An efficient reconfigurable architecture by characterizing most frequent logic functions. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–6.
[11] Ibrahim Ahmed, Linda L. Shen, and Vaughn Betz. 2019. Becoming More Tolerant: Designing FPGAs for Variable Supply Voltage. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*.
[12] Altera 2016. *Stratix IV Device Handbook*. Altera.
[13] Altera Corporation 2015. *Stratix V Device Handbook*. Altera Corporation. SV5V1.
[14] M. An, J. G. Steffan, and V. Betz. 2014. Speeding Up FPGA Placement: Parallel Algorithms and Methods. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 178–185.
[15] V. Betz and J. Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*. 213–222.
[16] Vaughn Betz and Jonathan Rose. 2000. Automatic Generation of FPGA Routing Architectures from High-level Descriptions. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 175–184.
[17] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers.
[18] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40.
[19] Compute Canada. 2019. www.computecanada.ca.

Table 27. VTR 8.0: VTR Benchmarks (Relative to VTR 7)

Benchmark	Netlist Primitives		Clocks	Logic Depth		IOs		CLBs		DSPs		RAMs		W_{min}		Routed Wirelength ($1.3 \cdot W_{min}$)		Routed CPD ($1.3 \cdot W_{min}$)		Odin II Time		ABC Time		Pack Time		Place Time		Route Time (Find W_{min})		Route Time ($1.3 \cdot W_{min}$)		VTR Flow Elapsed Time		VTR Flow Elapsed Time (excl. Find W_{min})		Peak Memory		
mcml	165,809	(0.86×)	1	27	(0.96×)	36	(1.00×)	7,196	(0.94×)	27	(0.90×)	159	(1.00×)	128	(0.98×)	1,007,246	(0.63×)	45.83	(1.03×)	24.94	(1.81×)	3,172.5	(17.49×)	204.7	(0.44×)	679.4	(0.80×)	1,321.6	(0.12×)	91.2	(0.38×)	5,597.2	(0.42×)	4,275.6	(2.33×)	1.93	(0.36×)	
LU32PEEng	101,542	(0.78×)	1	99	(0.95×)	114	(1.00×)	6,919	(0.79×)	32	(1.00×)	167	(0.99×)	136	(0.96×)	1,275,909	(0.72×)	75.25	(0.94×)	29.82	(4.25×)	853.4	(13.69×)	222.5	(0.83×)	536.1	(0.79×)	1,673.7	(0.02×)	146.9	(0.56×)	3,549.4	(0.04×)	1,875.7	(1.41×)	1.66	(0.28×)	
stereovision2	42,077	(0.93×)	1	3	(1.00×)	149	(1.00×)	1,703	(0.75×)	202	(0.95×)			88	(0.71×)	417,138	(0.57×)	14.25	(0.88×)	1.01	(1.09×)	6.6	(0.68×)	24.2	(0.78×)	88.7	(0.71×)	321.8	(0.03×)	11.4	(0.16×)	484.6	(0.05×)	162.8	(0.63×)	0.89	(0.49×)	
LU8PEEng	31,396	(0.79×)	1	103	(0.99×)	114	(1.00×)	2,062	(0.80×)	8	(1.00×)	44	(0.98×)	94	(0.98×)	295,304	(0.69×)	77.62	(0.89×)	2.44	(1.30×)	73.5	(4.46×)	64.2	(0.90×)	68.6	(0.92×)	177.5	(0.11×)	17.9	(0.34×)	427.7	(0.23×)	250.2	(1.09×)	0.53	(0.31×)	
bgm	24,865	(0.43×)	1	14	(0.52×)	257	(1.00×)	2,136	(0.54×)	11	(1.00×)			74	(0.90×)	281,735	(0.46×)	19.25	(0.62×)	9.07	(2.30×)	146.4	(6.87×)	39.5	(0.52×)	58.1	(0.33×)	101.8	(0.07×)	8.2	(0.25×)	388.3	(0.22×)	286.6	(0.88×)	0.46	(0.18×)	
stereovision0	21,789	(0.70×)	1	5	(0.83×)	157	(0.93×)	704	(0.63×)					56	(0.88×)	59,060	(0.50×)	3.54	(0.79×)	0.91	(1.00×)	8.3	(1.48×)	7.6	(0.37×)	8.6	(0.33×)	11.9	(0.18×)	1.3	(0.30×)	47.5	(0.36×)	35.6	(0.54×)	0.20	(0.26×)	
stereovision1	19,549	(0.71×)	1	3	(1.00×)	115	(0.86×)	702	(0.65×)	40	(1.05×)			78	(0.72×)	115,572	(0.56×)	5.44	(0.92×)	1.02	(1.01×)	40.0	(7.36×)	7.3	(0.34×)	11.1	(0.39×)	60.1	(0.07×)	3.8	(0.25×)	134.5	(0.15×)	74.4	(0.92×)	0.24	(0.33×)	
arm_core	18,437		1	18		133		1,034				40		116		212,114		20.97		1.11		65.6		29.6		23.0		170.7		14.4		340.5		169.8		0.36		○
blob_merge	11,415	(0.85×)	1	5	(1.00×)	36	(1.00×)	623	(0.88×)					62	(0.74×)	68,514	(0.60×)	14.95	(1.00×)	0.30	(1.21×)	36.0	(9.09×)	14.9	(0.78×)	7.8	(0.57×)	19.4	(0.21×)	1.3	(0.31×)	86.3	(0.63×)	66.9	(1.49×)	0.14	(0.31×)	
or1200	4,530	(0.82×)	1	8	(1.00×)	385	(1.00×)	255	(0.88×)	1	(1.00×)	2	(1.00×)	94	(1.15×)	45,258	(0.81×)	9.06	(1.01×)	0.24	(1.03×)	3.9	(2.52×)	6.4	(0.88×)	5.6	(0.87×)	10.2	(0.06×)	2.7	(1.23×)	32.6	(0.18×)	22.4	(1.15×)	0.09	(0.43×)	
mkDelayWorker32B	4,145	(0.36×)	1	5	(0.71×)	506	(0.99×)	505	(0.99×)			44	(1.02×)	40	(0.53×)	18,099	(0.14×)	6.52	(0.86×)	0.60	(1.25×)	5.4	(1.61×)	3.0	(0.23×)	10.1	(0.60×)	6.7	(0.00×)	0.9	(0.07×)	32.1	(0.02×)	25.4	(0.52×)	0.25	(0.49×)	
raygentop	2,934	(0.61×)	1	3	(0.75×)	214	(0.90×)	110	(0.53×)	8	(1.14×)			56	(0.74×)	20,468	(0.62×)	4.89	(0.97×)	0.19	(1.10×)	1.0	(1.20×)	1.9	(0.33×)	2.0	(0.53×)	7.0	(0.27×)	0.8	(0.44×)	14.9	(0.38×)	7.9	(0.57×)	0.06	(0.39×)	
mkSMAdapter4B	2,852	(0.69×)	1	4	(0.67×)	193	(0.99×)	199	(0.99×)			5	(1.00×)	50	(0.83×)	17,952	(0.69×)	5.33	(0.93×)	0.20	(1.22×)	1.6	(1.57×)	2.7	(0.48×)	2.3	(0.73×)	5.9	(0.35×)	0.3	(0.42×)	15.0	(0.52×)	9.1	(0.76×)	0.06	(0.39×)	
sha	2,744	(0.83×)	1	3	(0.60×)	38	(1.00×)	154	(0.76×)					74	(1.19×)	15,641	(0.86×)	10.13	(1.01×)	0.56	(1.74×)	245.3	(67.95×)	1.8	(0.60×)	1.5	(0.63×)	6.0	(0.19×)	0.3	(0.46×)	260.1	(6.10×)	254.1	(22.45×)	0.06	(0.43×)	
spree	1,229		1	15		45		65		1		3		70		11,210		10.52		0.12		0.7		2.0		0.8		2.4		0.3		7.5		5.1		0.04		○
mkPktMerge	1,160	(0.88×)	1	2	(0.67×)	311	(1.00×)	29	(1.38×)			15	(1.00×)	40	(0.80×)	13,486	(0.90×)	4.51	(1.09×)	0.10	(1.11×)	0.1	(0.60×)	0.5	(0.79×)	1.5	(0.68×)	2.6	(0.08×)	0.5	(0.20×)	6.9	(0.18×)	4.3	(0.69×)	0.06	(0.69×)	
boundtop	1,141	(0.20×)	1	3	(0.38×)	142	(0.52×)	94	(0.37×)					48	(0.89×)	3,152	(0.11×)	3.38	(0.54×)	0.25	(1.06×)	0.4	(0.28×)	0.7	(0.09×)	0.8	(0.19×)	0.9	(0.11×)	0.1	(0.08×)	4.9	(0.20×)	3.9	(0.25×)	0.04	(0.22×)	
diffeq1	886	(0.87×)	1	6	(1.00×)	162	(1.00×)	32	(0.91×)	5	(1.00×)			50	(0.81×)	8,945	(0.86×)	17.02	(1.01×)	0.03	(1.24×)	0.2	(0.79×)	0.3	(0.46×)	0.7	(0.90×)	6.3	(0.62×)	0.3	(0.42×)	8.7	(0.65×)	2.4	(0.74×)	0.04	(0.91×)	
diffeq2	599	(0.81×)	1	6	(1.20×)	66	(1.00×)	22	(0.85×)	5	(1.00×)			52	(0.90×)	7,411	(0.83×)	12.97	(1.02×)	0.02	(1.89×)	0.1	(0.80×)	0.4	(0.53×)	0.6	(0.76×)	2.3	(0.21×)	0.3	(0.63×)	4.5	(0.34×)	2.2	(0.87×)	0.04	(0.97×)	
ch_intrinsics	493	(0.55×)	1	3	(0.75×)	99	(1.00×)	64	(1.73×)					54	(1.13×)	1,304	(0.33×)	2.58	(0.68×)	0.06	(1.58×)	0.2	(0.59×)	0.1	(0.20×)	0.3	(0.81×)	0.6	(0.64×)	0.0	(0.35×)	2.1	(0.67×)	1.5	(0.68×)	0.03	(0.91×)	
stereovision3	321	(0.92×)	2	5	(1.25×)	11	(1.00×)	14	(1.08×)					30	(0.94×)	791	(1.06×)	2.65	(0.95×)	0.04	(1.28×)	0.1	(0.92×)	0.3	(1.02×)	0.1	(0.60×)	0.3	(0.98×)	0.0	(0.88×)	1.6	(1.21×)	1.3	(1.29×)	0.02	(1.30×)	
GEOMEAN	5,450.6	(0.68×)	1	7.2	(0.82×)	112.1	(0.95×)	291.6	(0.82×)	10.2	(1.00×)	15.7	(1.00×)	65.7	(0.87×)	36,594.1	(0.56×)	10.25	(0.89×)	0.46	(1.39×)	5.7	(2.39×)	4.4	(0.48×)	5.8	(0.60×)	15.4	(0.13×)	1.3	(0.33×)	50.6	(0.29×)	31.3	(0.99×)	0.14	(0.44×)	
GEOMEAN (> 10K)	33,242.9	(0.74×)	1	13.6	(0.89×)	105	(0.97×)	1,700.5	(0.74×)	29.2	(0.98×)	82.7	(0.99×)	88.6	(0.85×)	254,151.9	(0.59×)	19.68	(0.88×)	2.58	(1.53×)	81.1	(5.15×)	34.4	(0.58×)	50.7	(0.56×)	146.2	(0.08×)	10.8	(0.30×)	406.1	(0.19×)	241.7	(1.05×)	0.49	(0.30×)	
% TOTAL (> 10K)																				0.64%		39.1%		5.7%		13.4%		34.9%		2.7%		100.0%		64.9%				

Run-time in Seconds, Memory in GiB, WL in grid tiles, CPD in ns
○ VTR 7 error

Table 28. VPR 8.0: Titan v1.1.0 Benchmarks (Relative to VPR 7+)

Benchmark	Netlist Primitives	Clocks	IOs	LABs		DSPs		M9Ks		M144Ks		Device Grid Tiles	Routed Wirelength		Routed CPD (geomean)		Pack Time		Place Time		Route Time		Misc. Time		Total Time		Peak Memory	
gaussianblur	1,860,045	1	558	105,457		4		12		0		117,410					126.1	(0.76×)	576.8								■	○
bitcoin_miner	1,062,214	2	385	29,533	(1.02×)			1,668	(1.14×)	0	(0.00×)	43,139	12,604,393		1.01		12.1	(0.23×)	169.2	(1.40×)	23.8		12.3		217.4		12.28		□	
directrf	934,809	2	319	58,474		252		2,535		0		73,162					23.5		217.6								▲	○
sparcT1_chip2	766,880	1	1,890	29,716	(0.96×)	3	(1.00×)	580	(1.09×)			55,622	7,207,537	(0.61×)	19.93	(0.89×)	25.1	(0.47×)	88.2	(0.94×)	7.8	(0.02×)	14.3	(5.20×)	135.4	(0.23×)	11.41	(0.24×)		
LU_Network	630,376	19	405	31,026		112		1,175		0		35,478					11.8		86.0								■	○
LU230	568,367	2	373	17,049	(1.11×)	116	(1.00×)	5,040	(1.09×)	16	(0.05×)	135,676	18,171,449		10.13		20.6	(0.51×)	62.9		20.8		32.7		137.0		19.67			○
mes_noc	549,051	9	5	25,570	(1.03×)			800	(1.00×)			29,106	5,093,677	(0.56×)	9.90		24.8	(0.73×)	77.9	(0.85×)	9.8	(0.03×)	8.0	(3.59×)	120.5	(0.24×)	8.15	(0.21×)		
gsm_switch	491,989	4	138	20,827	(1.08×)			1,848	(1.15×)	0	(0.00×)	47,124	6,566,158	(0.70×)	5.59	(0.85×)	12.7	(0.50×)	39.3	(0.83×)	5.6	(0.00×)	11.8	(5.48×)	69.4	(0.02×)	9.17	(0.14×)		
denoise	343,755	1	852	18,202	(1.05×)	48	(1.00×)	377	(1.03×)			21,125	3,897,065	(0.66×)	864.44	(0.93×)	8.4	(0.38×)	79.3	(1.09×)	7.5	(0.01×)	6.0	(4.10×)	101.2	(0.10×)	6.17	(0.24×)		
sparcT2_core	288,458	1	451	14,215	(1.09×)			266	(1.00×)			16,280	4,604,317	(0.84×)	18.81	(1.53×)	11.2	(0.61×)	41.0	(1.29×)	6.4	(0.04×)	4.4	(4.40×)	63.0	(0.31×)	4.62	(0.25×)		
cholesky_bdti	256,234	1	162	9,678	(1.04×)	132	(1.00×)	600	(1.00×)			20,832	2,562,018	(0.66×)	9.34	(0.77×)	4.8	(0.61×)	14.2	(0.74×)	4.5	(0.04×)	5.1	(4.55×)	28.6	(0.20×)	5.02	(0.20×)		
minres	252,687	2	229	7,693	(1.07×)	149	(1.17×)	1,458	(1.27×)	0	(0.00×)	36,244	2,749,531	(0.61×)	6.28	(0.86×)	5.4	(0.64×)	13.5	(0.96×)	3.2	(0.03×)	8.3	(6.61×)	30.3	(0.21×)	6.61	(0.15×)		
stap_qrd	237,347	1	150	16,067	(1.04×)	74	(1.00×)	553	(1.31×)	0	(0.00×)	18,212	2,635,810	(0.62×)	8.06	(0.82×)	4.3	(0.56×)	25.7	(0.86×)	4.1	(0.10×)	4.7	(4.09×)	38.9	(0.47×)	4.52	(0.19×)		
openCV	212,826	1	208	7,135	(1.10×)	248	(1.47×)	787	(1.01×)	40	(0.83×)	38,930	3,271,085		11.50		5.3	(0.64×)	12.1	(0.76×)	6.5		8.9		32.9		6.65			△
dart	202,439	1	69	6,898	(1.07×)			530	(1.07×)	0	(0.00×)	13,837	2,533,682		13.32		6.6	(0.79×)	11.4	(0.92×)	3.5		3.6		25.1		3.58			△
bitonic_mesh	191,783	1	119	7,671	(1.12×)	169	(1.84×)	1,664	(1.07×)	0	(0.00×)	43,139	4,139,332	(0.53×)	15.33	(0.81×)	8.8	(0.78×)	17.9	(0.94×)	5.9	(0.00×)	9.9	(7.39×)	42.4	(0.03×)	7.05	(0.13×)		
segmentation	168,365	1	441	8,641	(1.06×)	21	(1.00×)	488	(1.33×)	0	(0.00×)	13,500	2,060,810	(0.67×)	880.13	(0.96×)	3.8	(0.43×)	23.2	(0.98×)	3.2	(0.01×)	3.5	(4.27×)	33.7	(0.09×)	3.55	(0.20×)		
SLAM_spheric	125,679	1	479	6,657	(1.04×)	66	(1.06×)	87	(1.00×)			10,890	2,116,663		82.97		3.7	(0.59×)	12.6	(0.95×)	3.7		2.7		22.7		2.99			□
des90	109,928	1	117	4,252	(1.07×)	88	(1.80×)	860	(1.05×)	0	(0.00×)	21,125	2,192,517	(0.59×)	12.50	(0.80×)	4.7	(0.85×)	8.4	(0.91×)	3.6	(0.02×)	4.8	(6.03×)	21.6	(0.09×)	3.92	(0.14×)		
cholesky_mc	108,499	1	262	4,792	(1.05×)	58	(1.00×)	444	(1.09×)	16	(0.80×)	11,193	1,203,602	(0.78×)	8.37	(0.95×)	1.9	(0.74×)	4.4	(0.82×)	2.2	(0.11×)	2.7	(4.11×)	11.3	(0.40×)	2.75	(0.17×)		
stereo_vision	93,171	3	506	3,038	(1.12×)	80	(1.05×)	135	(1.00×)			12,160	702,858	(0.73×)	3.19	(0.88×)	3.7	(2.96×)	2.3	(0.67×)	1.1	(0.19×)	2.7	(5.90×)	9.8	(0.89×)	2.68	(0.29×)		
sparcT1_core	91,580	1	310	3,910	(1.04×)	1	(1.00×)	131	(1.00×)			4,661	1,255,399	(0.78×)	8.50	(0.84×)	4.4	(1.02×)	4.7	(0.94×)	1.7	(0.08×)	1.3	(2.93×)	12.1	(0.40×)	1.90	(0.26×)		
neuron	90,857	1	77	3,310	(1.05×)	97	(1.04×)	185	(1.00×)			14,076	873,185	(0.69×)	6.07	(0.80×)	1.6	(0.88×)	2.7	(0.80×)	1.5	(0.13×)	3.3	(7.01×)	9.1	(0.51×)	2.94	(0.29×)		
GEOMEAN	286,921	1.6	236.2	12,112.4	(1.06×)	51.4	(1.13×)	530.2	(1.08×)	21.7	(0.31×)	26,191.7	3,071,830.2	(0.66×)	14.80	(0.89×)	8.1	(0.65×)	26.1	(0.92×)	4.6	(0.03×)	5.7	(4.88×)	38.5	(0.19×)	5.25	(0.20×)		

Run-time in Minutes, Memory in GiB, WL in grid tiles, CPD in ns
▲ VPR 8 time-out; ■ VPR 8 unroute; △ VPR 7+ time-out; □ VPR 7+ unroute; ○ VPR 7+ error

Table 29. VPR 8.0 High Effort: Titan v1.3.0 (Relative to Quartus 18.0)

Benchmark	Netlist Primitives	Clocks	IOs		LABs		DSPs		M9Ks		M144Ks		Device Grid Tiles		Limiting Resource	Routed Wirelength		Routed CPD (geomean)		Pack Time		Place Time		Route Time		Misc. Time		Total Time		STA Time		Peak Memory		
gaussianblur	1,859,014	1	558	(0.81×)	103,802	(1.12×)	2		12	(1.00×)			116,718		LAB	30,738,160		929.24		126.7		1,200.7		77.1		33.5		1,438.0		22.6		27.07		†
bitcoin_miner	1,087,537	1	385	(0.71×)	32,382	(1.00×)			1,331	(1.02×)			37,184	(1.03×)	LAB	10,238,973	(1.06×)	8.38	(0.93×)	12.3	(0.31×)	428.3	(0.97×)	10.0	(0.09×)	11.0	(0.24×)	461.6	(0.73×)	9.8	(0.44×)	11.39	(0.78×)	
directrf	930,989	2	319	(0.98×)	60,824	(1.75×)	240		2,535	(1.23×)	0	(0.00×)	74,495		M9K	12,269,002		9.71		17.3		418.3		84.2		20.9		540.7		10.9		16.95		†
sparcT1_chip2	760,412	21	1,891	(1.70×)	33,405	(4.73×)	3	(1.00×)	506	(1.00×)			57,753	(1.59×)	IO	7,706,606		5.00		27.2	(0.53×)	221.9	(0.85×)	6.2		15.9		271.2		6.9		11.22		‡
LU_Network	630,079	4	411	(2.04×)	31,050	(1.47×)	112	(1.00×)	1,175	(1.26×)	0	(0.00×)	35,860	(1.50×)	LAB	5,652,344	(1.19×)	5.05	(0.92×)	12.9	(0.69×)	188.9	(1.37×)	6.1	(0.20×)	10.3	(0.36×)	218.2	(1.01×)	8.1	(0.22×)	10.03	(0.96×)	
LU230	568,001	2	373	(1.00×)	16,619	(0.51×)	116	(1.00×)	5,040	(4.05×)	16	(1.00×)	137,170	(3.78×)	M9K	15,448,573	(1.45×)	10.02	(1.03×)	21.6	(0.75×)	200.4	(1.07×)	183.4	(0.67×)	33.9	(0.64×)	439.3	(0.81×)	8.1	(0.25×)	17.10	(1.22×)	
mes_noc	547,568	9	5	(1.00×)	24,167	(0.81×)			800	(1.19×)			27,936	(0.77×)	LAB	4,942,841	(1.03×)	8.38	(1.16×)	23.2	(1.51×)	158.1	(0.87×)	7.7	(0.26×)	8.1	(0.28×)	197.1	(0.77×)	6.1	(0.21×)	7.79	(0.88×)	
gsm_switch	490,070	3	138	(1.01×)	21,534	(0.72×)			1,848	(1.15×)	0	(0.00×)	48,195	(1.33×)	M9K	5,314,814	(0.95×)	6.10	(1.25×)	12.7	(1.16×)	110.9	(0.64×)	17.1	(0.35×)	12.2	(0.49×)	152.9	(0.59×)	4.3	(0.27×)	8.76	(1.01×)	
sparcT2_core	300,220	1	451	(0.95×)	14,663	(1.04×)			260	(1.00×)			17,176	(0.72×)	LAB	4,562,316	(1.06×)	10.83	(1.09×)	11.7	(1.89×)	83.3	(1.44×)	6.3	(0.18×)	4.8	(0.36×)	106.1	(0.94×)	3.4	(0.33×)	4.68	(0.88×)	
denoise	274,786	1	852	(0.87×)	14,434	(0.79×)	24	(0.73×)	359	(2.74×)			16,912	(0.47×)	LAB	3,067,873	(1.10×)	851.00	(0.90×)	6.6	(1.13×)	115.3	(0.65×)	4.5	(0.19×)	4.7	(0.43×)	131.2	(0.60×)	3.6	(0.53×)	5.24	(0.88×)	
minres	257,480	2	229	(0.67×)	8,030	(0.63×)	78	(0.93×)	1,458	(1.31×)			37,184	(1.56×)	M9K	2,708,510	(1.46×)	4.16	(1.12×)	6.0	(1.03×)	54.6	(1.32×)	3.2	(0.38×)	9.2	(0.67×)	73.0	(1.05×)	2.5	(0.22×)	6.38	(1.12×)	
cholesky_bdti	255,478	1	162	(1.00×)	9,725	(1.09×)	132	(1.00×)	600	(1.00×)			21,125	(1.85×)	DSP	2,572,119	(1.48×)	9.09	(1.68×)	4.7	(0.97×)	41.1	(0.91×)	8.2	(1.01×)	5.7	(0.55×)	59.8	(0.87×)	2.3	(0.22×)	5.19	(1.09×)	
stap_qrd	234,177	1	150	(0.95×)	16,029	(1.82×)	74	(1.00×)	553	(1.00×)			18,762	(1.64×)	LAB	2,595,613	(1.53×)	7.67	(2.04×)	4.5	(0.76×)	64.3	(1.65×)	5.5	(0.97×)	4.9	(0.55×)	79.3	(1.33×)	2.8	(0.39×)	4.54	(1.02×)	
openCV	212,553	1	208	(0.62×)	7,128	(0.84×)	213	(1.68×)	785	(1.17×)	40	(0.83×)	32,395	(1.36×)	DSP	2,997,630	(1.46×)	11.20	(1.24×)	5.3	(1.55×)	42.2	(1.17×)	6.3	(0.99×)	8.3	(0.98×)	62.1	(1.14×)	2.5	(0.38×)	5.69	(1.13×)	
dart	202,402	1	69	(1.00×)	7,325	(0.83×)			530	(1.00×)			14,076	(1.23×)	M9K	2,117,081	(1.02×)	11.93	(1.29×)	6.5	(1.61×)	29.4	(0.82×)	2.0	(0.14×)	3.8	(0.11×)	41.7	(0.48×)	2.2	(0.26×)	3.67	(0.89×)	
bitonic_mesh	190,746	1	119	(0.88×)	7,381	(0.50×)	85	(0.71×)	1,664	(1.03×)	0	(0.00×)	43,318	(1.20×)	M9K	4,258,536	(0.80×)	13.74	(1.32×)	8.3	(2.28×)	68.7	(1.42×)	5.5	(0.15×)	10.8	(0.59×)	93.4	(0.88×)	3.4	(0.24×)	6.73	(1.13×)	
segmentation	137,832	1	441	(0.82×)	7,093	(0.78×)	15	(0.83×)	481	(1.87×)	0	(0.00×)	13,736	(0.58×)	M9K	1,704,886	(1.19×)	848.22	(0.92×)	3.2	(1.33×)	39.0	(0.66×)	2.5	(0.23×)	3.8	(0.68×)	48.5	(0.62×)	1.7	(0.52×)	3.36	(0.90×)	
SLAM_spheric	111,354	1	479	(0.79×)	5,607	(0.68×)	37	(0.56×)					6,984	(0.29×)	LAB	1,671,905	(1.15×)	79.06	(1.11×)	3.2	(1.80×)	20.4	(0.73×)	3.1	(0.54×)	2.0	(0.38×)	28.8	(0.70×)	1.4	(0.36×)	2.55	(0.78×)	
des90	110,549	1	117	(0.88×)	4,248	(0.75×)	44	(0.50×)	860	(1.00×)			21,717	(1.90×)	M9K	2,147,716	(1.26×)	12.71	(1.60×)	4.5	(2.70×)	26.7	(1.33×)	2.7	(0.44×)	5.8	(0.69×)	39.6	(1.10×)	1.8	(0.25×)	3.91	(1.08×)	
cholesky_mc	108,592	1	262	(0.98×)	4,824	(0.88×)	59	(1.00×)	444	(1.19×)	16	(0.80×)	11,625	(1.02×)	M9K	1,235,099	(1.96×)	7.26	(1.58×)	2.0	(1.15×)	13.7	(1.17×)	2.8	(0.98×)	3.1	(0.68×)	21.5	(1.04×)	1.2	(0.30×)	2.92	(0.92×)	
stereo_vision	94,090	2	506	(1.30×)	3,327	(0.87×)	76	(4.00×)	113	(1.00×)			12,384	(2.16×)	DSP	608,530	(1.52×)	3.47	(0.96×)	1.2	(0.92×)	8.9	(0.86×)	0.6	(0.38×)	3.1	(1.07×)	13.8	(0.86×)	0.6	(0.32×)	2.47	(0.99×)	
sparcT1_core	91,975	1	310	(0.94×)	3,983	(0.95×)	1	(1.00×)	128	(1.01×)			5,002	(0.87×)	LAB	1,225,016	(1.00×)	8.67	(1.13×)	4.0	(3.44×)	10.0	(0.83×)	1.3	(0.20×)	1.5	(0.38×)	16.9	(0.70×)	1.0	(0.28×)	1.98	(0.80×)	
neuron	86,875	1	77	(0.74×)	3,136	(0.68×)	89	(0.81×)	136	(4.00×)	0	(0.00×)	12,384	(1.08×)	DSP	765,195	(2.27×)	5.56	(1.32×)	1.4	(0.98×)	8.9	(1.05×)	0.9	(0.35×)	3.2	(0.84×)	14.5	(0.88×)	0.8	(0.20×)	2.75	(0.94×)	
GEOMEAN	280,371.8	1.6	236.4	(0.95×)	11,977.2	(0.95×)	39.7	(0.97×)	551.1	(1.30×)	21.7	(0.87×)	25,342.2	(1.15×)		3,427,344.9	(1.26×)	16.00	(1.20×)	7.4	(1.18×)	66.6	(1.00×)	6.0	(0.34×)	6.9	(0.49×)	91.6	(0.83×)	3.2	(0.30×)	5.86	(0.96×)	
% TOTAL																				7.2%		78.0%		9.8%		4.9%		100.0%		2.4%				

Run-time in Minutes, Memory in GiB, WL in grid tiles, CPD in ns
STA Time is included in Pack/Place/Route/Misc. and Total Times
† Quartus device size exceeded; ‡ Quartus unroute

[20] Yao-Wen Chang, D. F. Wong, and C. K. Wong. 1996. Universal Switch Modules for FPGA Design. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (Jan. 1996), 80–101.
[21] Baudouin Chauviere, Aurlien Alacchi, Edouard Giacomin, Xifan Tang, and Pierre-Emmanuel Gaillardon. 2019. OpenFPGA: Complete Open Source Framework for FPGA Prototyping.
[22] Deming Chen, Jason Cong, and Peichen Pan. 2006. FPGA Design Automation: A Survey. *Found. Trends Electron. Des. Autom.* 1, 3 (Jan. 2006), 139–169.
[23] Doris Chen, Deshanand Singh, Jeffrey Chromczak, David Lewis, Ryan Fung, et al. 2010. A Comprehensive Approach to Modeling, Characterizing and Optimizing for Metastability in FPGAs. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 167–176.
[24] D. T. Chen, K. Vorwerk, and A. Kennings. 2007. Improving Timing-Driven FPGA Packing with Physical Information. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 117–123.
[25] Gang Chen and Jason Cong. 2004. Simultaneous Timing Driven Clustering and Placement for FPGAs. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 158–167.
[26] S. Chen and Y. Chang. 2015. Routing-architecture-aware analytical placement for heterogeneous FPGAs. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.
[27] Y. Chen, S. Chen, and Y. Chang. 2014. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *Int. Conf. on Computer-Aided Design (ICCAD)*. 647–654.
[28] C. Chiasson and V. Betz. 2013. COFFE: Fully-automated transistor sizing for FPGAs. In *Int. Conf. on Field-Programmable Technology (FPT)*. 34–41.
[29] S. A. Chin, J. Luu, S. Huda, and J. H. Anderson. 2016. Hybrid LUT/Multiplexer FPGA Logic Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 4 (April 2016), 1280–1292.
[30] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (Mar 2018), 8–20.
[31] Intel Corporation. 2019. Quartus. https://www.intel.ca/content/www/ca/en/software/programmable/quartus-/prime/overview.html
[32] Xilinx Corporation. 2019. Vivado. https://www.xilinx.com/products/design-/tools/vivado.html
[33] DARPA. 2016. Reconfigurable Imaging (ReImagine). https://www.darpa.mil/attachments/Final_Compiled_ReImagineProposersDay.pdf
[34] André DeHon. 1999. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, Why You Don't Really Want 100% LUT Utilization). In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 69–78.
[35] C. Ebeling et al. 2016. Stratix™10 High Performance Routable Clock Networks. In *Int. Symp. on FPGAs*. 64–73.
[36] Z. Ebrahimi, B. Khaleghi, and H. Asadi. 2017. PEAF: A Power-Efficient Architecture for SRAM-Based FPGAs Using Reconfigurable Hard Logic Design in Dark Silicon Era. *IEEE Trans. Comput.* 66, 6 (June 2017), 982–995.
[37] B. Erbagci, N. E. Can Akkaya, C. Erbagci, and K. Mai. 2019. An Inherently Secure FPGA using PUF Hardware-Entanglement and Side-Channel Resistant Logic in 65nm Bulk CMOS. In *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference (ESSCIRC)*. 65–68. https://doi.org/10.1109/ESSCIRC.2019.8902789
[38] Wenyi Feng, Jonathan Greene, Kristofer Vorwerk, Val Pevzner, and Arun Kundu. 2014. Rent's Rule Based FPGA Packing for Routability Optimization. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 31–34.
[39] C. Fobel, G. Grewal, and D. Stacey. 2014. A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and GPU architectures. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–8.
[40] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA) (FPGA '19)*. 84–93.
[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.
[42] Mingyu Gao, Christina Delimitrou, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, et al. 2016. DRAF: A Low-power DRAM-based Reconfigurable Acceleration Fabric. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 506–518.
[43] J. B. Goeders, G. G. F. Lemieux, and S. J. E. Wilton. 2011. Deterministic Timing-Driven Parallel Placement by Simulated Annealing Using Half-Box Window Decomposition. In *ReConFig*. 41–48.

[44] J. B. Goeders and S. J. E. Wilton. 2012. VersaPower: Power estimation for diverse FPGA architectures. In *Int. Conf. on Field-Programmable Technology (FPT)*. 229–234.
[45] Brett Grady and Jason H. Anderson. 2018. Synthesizable Verilog Backend for the VTR FPGA Evaluation Framework. In *Int. Conf. on Field-Programmable Technology (FPT)*.
[46] Travis Haroldsen, Brent Nelson, and Brad Hutchings. 2015. RapidSmith 2: A Framework for BEL-level CAD Exploration on Xilinx FPGAs. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 66–69.
[47] P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (July 1968), 100–107.
[48] K. Honda, T. Imagawa, and H. Ochi. 2017. Placement algorithm for mixed-grained reconfigurable architecture with dedicated carry chain. In *IEEE International System-on-Chip Conference (SOCC)*. 80–85.
[49] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A Parallel Deterministic Router Based on Spatial Partitioning and Scheduling. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 67–76.
[50] Chin Hau Hoo, A. Kumar, and Yajun Ha. 2015. ParaLaR: A parallel FPGA router based on Lagrangian relaxation. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–6.
[51] K. Huang, R. Zhao, W. He, and Y. Lian. 2016. High-Density and High-Reliability Nonvolatile Field-Programmable Gate Array With Stacked 1D2R RRAM Array. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 1 (Jan 2016), 139–150.
[52] Zhihong Huang, Xing Wei, Grace Zgheib, Wei Li, Yu Lin, et al. 2017. NAND-NOR: A Compact, Fast, and Delay Balanced FPGA Logic Element. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 135–140.
[53] S. Huda and J. H. Anderson. 2017. Leveraging Unused Resources for Energy Optimization of FPGA Interconnect. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 8 (Aug 2017), 2307–2320.
[54] E. Hung. 2015. Mind the (synthesis) gap: Examining where academic FPGA tools lag behind industry. In *nternational Conference on Field Programmable Logic and Applications (FPL)*. 1–4.
[55] E. Hung, F. Eslami, and S. J. E. Wilton. 2013. Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 45–52.
[56] E. Hung and S. J. E. Wilton. 2014. Incremental Trace-Buffer Insertion for FPGA Debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 4 (April 2014), 850–863.
[57] C. Huriaux, O. Sentieys, and R. Tessier. 2016. Effects of I/O routing through column interfaces in embedded FPGA fabrics. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–9.
[58] Mike Hutton, David Karchmer, Bryan Archell, and Jason Govig. 2005. Efficient Static Timing Analysis and Applications Using Edge Masks. In *ACM/SIGDA 13th Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 174–183.
[59] Xilinx Inc. 1994. The Programmable Logic Data Book.
[60] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon. 2010. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 149–156.
[61] Edin Kadric, David Lakata, and André DeHon. 2015. Impact of Memory Architecture on FPGA Energy Consumption. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 146–155.
[62] B. Khaleghi, B. Omidi, H. Amrouch, J. Henkel, and H. Asadi. 2016. Stress-aware routing to mitigate aging effects in SRAM-based FPGAs. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–8.
[63] F. F. Khan and A. Ye. 2016. An evaluation on the accuracy of the minimum width transistor area models in ranking the layout area of FPGA architectures. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–11.
[64] Jin Hee Kim and Jason H. Anderson. 2017. Synthesizable Standard Cell FPGA Fabrics Targetable by the Verilog-to-Routing CAD Flow. *ACM Trans. Reconfigurable Technol. Syst.* 10, 2, Article 11 (April 2017), 23 pages.
[65] N. Kulkarni, J. Yang, and S. Vrudhula. 2014. A fast, energy efficient, field programmable threshold-logic array. In *Int. Conf. on Field-Programmable Technology (FPT)*. 300–305.

[66] MIT Lincoln Lab. 2016. Reconfigurable Integrated Circuits for ReImagine. https://www.darpa.mil/attachments/MITLL_ProposerDaySlides%20v3.pdf

[67] C. Lavin. 2019. Building domain-specific implementation tools using the RapidWright framework. In *FPGA*.

[68] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, et al. 2011. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 349–355.

[69] J. Legault, P. Patros, and K. B. Kent. 2018. Towards Trainable Synthesis for Optimized Circuit Deployment on FPGA. In *2018 International Symposium on Rapid System Prototyping (RSP)*. 90–96. https://doi.org/10.1109/RSP.2018.8631999

[70] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, et al. 2003. The Stratix™Routing and Logic Architecture. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays (FPGA '03)*. ACM, New York, NY, USA, 12–20. https://doi.org/10.1145/611817.611821

[71] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, et al. 2013. Architectural Enhancements in Stratix V™. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2435264.2435292

[72] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, et al. 2016. The Stratix™10 Highly Pipelined FPGA Architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/2847263.2847267

[73] Ang Li and David Wentzlaff. 2019. PRGA: An Open-source Framework for Building and Using Custom FPGAs. In *Workshop on Open Source Design Automation*.

[74] Hao Jun Liu. 2014. *Archipelago - An Open Source FPGA with Toolflow Support*. Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-/2014-/43.html

[75] Jason Luu. 2014. *Architecture-aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays*. Master's thesis. University of Toronto. http://hdl.handle.net/1807/75854

[76] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, et al. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 2, Article 6 (July 2014), 30 pages.

[77] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, et al. 2011. VPR 5.0: FPGA CAD and Architecture Exploration Tools with Single-driver Routing, Heterogeneity and Process Scaling. *ACM Trans. Reconfigurable Technol. Syst.* 4, 4, Article 32 (Dec. 2011), 23 pages.

[78] J. Luu, C. McCullough, S. Wang, S. Huda, B. Yan, et al. 2014. On Hard Adders and Carry Chains in FPGAs. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 52–59.

[79] Jason Luu, Jonathan Rose, and Jason Anderson. 2014. Towards Interconnect-adaptive Packing for FPGAs. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 21–30.

[80] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A Negotiation-based Performance-driven Router for FPGAs. In *Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 111–117.

[81] Kevin E. Murray and Vaughn Betz. 2018. Tatum: Parallel Timing Analysis for Faster Design Cycles and Improved Optimization. In *Int. Conf. on Field-Programmable Technology (FPT)*.

[82] Kevin E. Murray and Vaughn Betz. 2019. Adaptive FPGA Placement Optimization via Reinforcement Learning. In *ACM/IEE Workshop on Machine Learning for CAD (MLCAD)*. 1–6.

[83] Kevin. E. Murray, J. Luu, M. J. P. Walker, C. McCullough, S. Wang, et al. 2020. Optimizing FPGA Logic Block Architectures for Arithmetic. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* (2020). To Appear.

[84] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2013. Titan: Enabling Large and Complex Benchmarks in Academic CAD. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*.

[85] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2015. Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD. *ACM Trans. Reconfigurable Technol. Syst.* 8, 2, Article 10 (March 2015), 18 pages.

[86] Kevin. E. Murray, Sheng Zhong, and Vaughn Betz. 2020. AIR: A Fast but Lazy Timing-Driven FPGA Router. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 1–7.

[87] E. Nasiri, J. Shaikh, A. Hahn Pereira, and V. Betz. 2016. Multiple Dice Working as One: CAD Flows and Routing Architectures for Silicon Interposer FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 5 (May 2016), 1821–1834.

[88] Xinyu Niu, Wayne Luk, and Yu Wang. 2015. EURECA: On-Chip Configuration Generation for Effective Dynamic Data Access. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 74–83.

[89] Hadi Parandeh-Afshar, Hind Benbihi, David Novo, and Paolo Ienne. 2012. Rethinking FPGAs: Elude the Flexibility Excess of LUTs with And-inverter Cones. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 119–128.

[90] M. Patrou, J. P. Legault, A. Graham, and K. B. Kent. 2019. Improving Digital Circuit Simulation with Batch-Parallel Logic Evaluation. In *To appear in Euromicro Digital System Design.*

[91] Oleg Petelin. 2016. *CAD Tools and Architectures for Improved FPGA Interconnect.* Master's thesis. University of Toronto. http://hdl.handle.net/1807/75854

[92] O. Petelin and V. Betz. 2016. The speed of diversity: Exploring complex FPGA routing topologies for the global metal layer. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–10.

[93] Andrew Putnam et al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ACM/IEEE Int. Symp. on Comput. Architecture (ISCA)*. 13–24.

[94] J. Richardson et al. 2010. Comparative Analysis of HPC and Accelerator Devices: Computation, Memory, I/O, and Power. In *Int. Workshop on High-Performance Reconfigurable Comput. Technol. and Appl. (HPRCTA)*. 1–10.

[95] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, et al. 2012. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 77–86.

[96] Raphael Y. Rubin and André M. DeHon. 2011. Timing-driven Pathfinder Pathology and Remediation: Quantifying and Reducing Delay Noise in VPR-pathfinder. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 173–176.

[97] Z. Seifoori, B. Khaleghi, and H. Asadi. 2017. A power gating switch box architecture in routing network of SRAM-based FPGAs in dark silicon era. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 1342–1347.

[98] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, et al. 2019. Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*.

[99] A. Sharma, S. Hauck, and C. Ebeling. 2005. Architecture-adaptive routability-driven placement for FPGAs. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 427–432.

[100] Minghua Shen and Guojie Luo. 2017. Corolla: GPU-Accelerated FPGA Routing Based on Subgraph Dynamic Expansion. In *FPGA*. 105–114.

[101] Amit Singh and Malgorzata Marek-Sadowska. 2002. Efficient Circuit Clustering for Area and Power Reduction in FPGAs. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 59–66.

[102] K. Siozios and D. Soudris. 2016. A Customizable Framework for Application Implementation onto 3-D FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 11 (Nov 2016), 1783–1796.

[103] Satish Sivaswamy, Gang Wang, Cristinel Ababei, Kia Bazargan, Ryan Kastner, et al. 2005. HARP: Hard-wired Routing Pattern FPGAs. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 21–29.

[104] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, et al. 2011. Torc: Towards an Open-source Tool Flow. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 41–44.

[105] M. Stojilovic. 2017. Parallel FPGA routing: Survey and challenges. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.23919/FPL.2017.8056782

[106] X. Sun, H. Zhou, and L. Wang. 2019. Bent Routing Pattern for FPGA. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 9–16. https://doi.org/10.1109/FPL.2019.00012

[107] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. 2019. Network-on-Chip Programmable Platform in VersalTM ACAP Architecture. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 212–221.

[108] Jordan S. Swartz, Vaughn Betz, and Jonathan Rose. 1998. A Fast Routability-driven Router for FPGAs. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. 140–149.

[109] Berkley Logic Synthesis and Verification Group. 2018. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/ Revision 1fc200ffacabed1796639b562181051614f5fedb.

[110] X. Tang, P. Gaillardon, and G. De Micheli. 2014. A high-performance low-power near-Vt RRAM-based FPGA. In *Int. Conf. on Field-Programmable Technology (FPT)*. 207–214.

[111] John Teifel, Matthew E. Land, and Russel. D. Miller. 2016. Improving ASIC Reuse with Embedded FPGA Fabrics. In *Government Microcircuit Applications & Critical Technology Conference*.

[112] J. Tian, G. R. Reddy, J. Wang, W. Swartz, Y. Makris, et al. 2017. A field programmable transistor array featuring single-cycle partial/full dynamic reconfiguration. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1336–1341. https://doi.org/10.23919/DATE.2017.7927200

[113] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. 2017. Liquid: High quality scalable placement for large heterogeneous FPGAs. In *Int. Conf. on Field Programmable Technology (FPT)*. 17–24.

[114] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. 2018. How Preserving Circuit Design Hierarchy During FPGA Packing Leads to Better Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 3 (March 2018), 629–642.

[115] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. 2019. CRoute: A Fast High-quality Timing-driven Connection-based FPGA Router. In *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*.

[116] M. Wainberg and V. Betz. 2015. Robust Optimization of Multiple Timing Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 12 (2015), 1942–1953.

[117] Chunan Wei, Ashutosh Dhar, and Deming Chen. 2015. A Scalable and High-density FPGA Architecture with Multi-level Phase Change Memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1365–1370.

[118] S. Wilton. 1997. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*. Ph.D. Dissertation. University of Toronto.

[119] Clifford Wolf. 2019. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/

[120] Xilinx Inc. 2016. *UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices*. Xilinx Inc. WP477 v1.0.

[121] Xilinx Inc. 2018. *Zynq UltraScale+ MPSoC Data Sheet*. Xilinx Inc. DS891 v1.7.

[122] Xilinx Inc. 2019. *Versal Architecture and Product Data Sheet*. Xilinx Inc. DS950 v1.1.

[123] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, et al. 2017. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. In *Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 157–166.

[124] S. Yang. 1991. *Logic Synthesis and Optimization Benchmarks User Guide 3.0*. Technical Report. MCNC.

[125] Sadegh Yazdanshenas and Vaughn Betz. 2019. COFFE 2: Automatic Modelling and Optimization of Complex and Heterogeneous FPGA Architectures. *ACM Trans. Reconfigurable Technol. Syst.* 12, 1, Article 3 (Jan. 2019), 27 pages.

[126] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. 2017. Don't Forget the Memory: Automatic Block RAM Modelling, Optimization, and Architecture Exploration. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. 115–124.

[127] Cunxi Yu and Zhiru Zhang. 2019. Painting on Placement: Forecasting Routing Congestion Using Conditional Generative Adversarial Nets. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC 19)*. Association for Computing Machinery, New York, NY, USA, Article Article 219, 6 pages. https://doi.org/10.1145/3316781.3317876

[128] G. Yu, T. Y. Cheng, B. Kettlewell, H. Liew, M. Seok, et al. 2017. FPGA with Improved Routability and Robustness in 130nm CMOS with Open-Source CAD Targetability. *ArXiv e-prints* (Dec. 2017). arXiv:1712.03411

[129] J. Yuan, L. Wang, X. Zhou, Y. Xia, and J. Hu. 2017. RBSA: Range-based simulated annealing for FPGA placement. In *Int. Conf. on Field Programmable Technology (FPT)*. 1–8.

[130] G. Zgheib and P. Ienne. 2017. Evaluating FPGA clusters under wide ranges of design parameters. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*. 1–8.

[131] Grace Zgheib, Liqun Yang, Zhihong Huang, David Novo, Hadi Parandeh-Afshar, et al. 2014. Revisiting And-inverter Cones. In *ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays (FPGA)*. 45–54.

Received July 2019; revised December 2019; revised January 2020; accepted March 2020.