VTR 9: Open-Source CAD for Fabric and Beyond FPGA Architecture Exploration

MOHAMED A. ELGAMMAL, University of Toronto, Toronto, Ontario, Canada and MangoBoost, Toronto, Ontario, Canada
AMIN MOHAGHEGH, SOHEIL GHOLAMI SHAHROUZ, FATEMEHSADAT MAHMOUDI, FAHRICAN KOŞAR, and KIMIA TALAEI, University of Toronto, Toronto, Ontario, Canada
JOSHUA FIFE and DANIEL KHADIVI, University of New Brunswick, Fredericton, New Brunswick, Canada
KEVIN MURRAY, Cerebras Systems, Toronto, Ontario, Canada
ANDREW BOUTROS, University of Waterloo, Waterloo, Ontario, Canada
KENNETH B. KENT and JEFF GOEDERS, University of New Brunswick, Fredericton, New Brunswick, Canada
VAUGHN BETZ, University of Toronto, Toronto, Ontario, Canada

This work details the capabilities of a major new release of the Verilog-to-Routing (VTR) open source FPGA CAD tool flow. Enhancements include generalizations of VTR's architecture modeling language and optimizers to enable a more diverse set of programmable routing fabrics, FPGAs with embedded hard Networks-on-Chip (NoCs) and three-dimensional 3D FPGA systems that leverage stacked silicon integration. The new Parmys logic synthesis flow improves language coverage and result quality, and the physical implementation flow includes a more efficient placement engine, floorplanning constraints to guide placement, the ability to perform single-stage (flat) routing to improve quality, and parallel routing algorithms to reduce CPU time. This release also includes new architecture captures of recent commercial devices (Xilinx's 7-series and Altera's Stratix 10) and new benchmark suites (Titanium25 and Hermes) to aid FPGA architecture investigation. Verilog language coverage is greatly improved with the new Parmys logic synthesis flow, enabling more designs to be used

We are grateful to funding support from NSERC, the Intel/VMware Crossroads Academic 3D FPGA Research Centre, QuickLogic, and Google.
Authors' Contact Information: Mohamed A. Elgammal, University of Toronto, Toronto, Ontario, Canada and MangoBoost, Toronto, Ontario, Canada; e-mail: mohamed.elgammal@mail.utoronto.ca; Amin Mohaghegh, University of Toronto, Toronto, Ontario, Canada; e-mail: amin.mohaghegh@mail.utoronto.ca; Soheil Gholami Shahrouz, University of Toronto, Toronto, Ontario, Canada; e-mail: s.shahrouz@mail.utoronto.ca; Fatemehsadat Mahmoudi, University of Toronto, Toronto, Ontario, Canada; e-mail: sara.mahmoudi@mail.utoronto.ca; Fahrican Koşar, University of Toronto, Toronto, Ontario, Canada; e-mail: f.kosar@mail.utoronto.ca; Kimia Talaei, University of Toronto, Toronto, Ontario, Canada; e-mail: kimia.talaei@mail.utoronto.ca; Joshua Fife, University of New Brunswick, Fredericton, New Brunswick, Canada; e-mail: chem3000@byu.edu; Daniel Khadivi, Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada; e-mail: d.khadivi@unb.ca; Kevin Murray, Cerebras Systems, Toronto, Ontario, Canada; e-mail: kevin.murray@cerebras.net; Andrew Boutros, University of Waterloo, Waterloo, Ontario, Canada; e-mail: andrew.boutros@uwaterloo.ca; Kenneth B. Kent, Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada; e-mail: ken@unb.ca; Jeff Goeders, University of New Brunswick, Fredericton, New Brunswick, Canada; e-mail: jgoeders@byu.edu; Vaughn Betz (corresponding author), University of Toronto, Toronto, Ontario, Canada; e-mail: vaughn@eecg.utoronto.edu.

ACM 1936-7414/2025/8-ART39
https://doi.org/10.1145/3734798

with VTR. Finally, the placement and routing engines have beeenbeen sped up by 4× and 2.2× vs. VTR 8, respectively, leading to an overall physical implementation flow CPU time reduction of 48% with better result quality on average compared to VTR 8.

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), Field Programmable Gate Array, Packing, Placement, Routing, Verilog To Routing (VTR)

ACM Reference format:
Mohamed A. Elgammal, Amin Mohaghegh, Soheil Gholami Shahrouz, Fatemehsadat Mahmoudi, Fahrican Koşar, Kimia Talaei, Joshua Fife, Daniel Khadivi, Kevin Murray, Andrew Boutros, Kenneth B. Kent, Jeff Goeders, and Vaughn Betz. 2025. VTR 9: Open-Source CAD for Fabric and Beyond FPGA Architecture Exploration. *ACM Trans. Reconfig. Technol. Syst.* 18, 3, Article 39 (August 2025), 53 pages.
https://doi.org/10.1145/3734798

1 Introduction

For many years, the performance of computing platforms improved from one generation to the next mainly by fabricating smaller, faster, and cheaper transistors following the trend predicted by Moore's Law [1]. More recently, the gains of process scaling have diminished due to fundamental challenges in manufacturing technologies and chip power dissipation as transistor dimensions approach the scale of a few silicon atoms [2]. On the other hand, there is increasing demand for higher compute performance due to the tremendous amount of data generated by mobile devices and the adoption of compute-intensive **Deep Learning (DL)** algorithms in numerous application domains both at the edge and in large-scale datacenters. As a result, specialized computing architectures are being widely deployed in production to deliver the required performance of critical high-demand applications [3, 4]. However, the design, manufacturing, testing, and deployment cycle of specialized **Application-Specific Integrated Circuit (ASIC)** chips is very costly and time consuming, making them suitable only for the largest markets or relatively stable applications.

Field-Programmable Gate Arrays (FPGAs) enable the implementation of specialized hardware at a fraction of the development time and cost of ASIC chips due to their hardware reconfigurability and diverse pre-fabricated high-speed external interfaces. A designer can describe any custom processing pipeline in a **Hardware Description Language (HDL)** (e.g., Verilog or VHDL), which is then synthesized, placed, and routed using a complex **Computer-Aided Design (CAD)** flow to generate a configuration bitstream that is used to program an off-the-shelf FPGA to implement the desired functionality. The FPGA can then be reconfigured *in-field* to implement a different functionality in hardware as the application design evolves over time. FPGAs have been deployed as datacenter accelerators [5, 6] and used to build custom processing pipelines for a wide variety of applications such as wireless communications [7], networking infrastructure [8–10], high-frequency trading [11], DL inference [12–14], genomics [15], and biophotonic cancer treatment simulation [16].

Over the past three decades, FPGAs have been growing in size and complexity to enhance their efficiency across a diverse range of application domains [17]. For example, the AMD Versal Premium devices contain up to 18.5 million logic cells [18] and recent devices from Intel embed specialized tensor cores in the FPGA fabric to significantly enhance their arithmetic density for DL applications [19]. The continuous growth in size and complexity of modern FPGA architectures and the application designs implemented on them requires continuous algorithmic innovation to develop *scalable* CAD tools that can achieve *high* **Quality of Results (QoR)** while maintaining reasonable bitstream compilation runtime. Additionally, CAD tools are also essential for quantitatively evaluating new FPGA architectural ideas. This requires a *flexible* and *data-driven* flow that accepts as an input not only the application designs to be mapped to an FPGA but also a detailed description of the target FPGA architecture. Such a flexible flow enables architects to experiment with novel ideas on both the architecture (e.g., organization of existing FPGA components [20, 21] or new embedded hard blocks [22, 23]) and microarchitecture levels (e.g., new **Logic Blocks [LBs]** [24, 25] or alternative circuit-level implementations of routing multiplexers [26]), and then compare different ideas based on application design implementation quality metrics such as **Critical Path Delay (CPD)**, routed **Wirelength (WL)**, and resource utilization.

While commercial closed-source FPGA CAD tools such as the Intel Quartus and AMD Vivado suites are continuously evolving to improve scalability and implementation QoR, they cannot be used for architecture exploration since they can only map designs to a pre-defined set of vendor-specific FPGA families. To this end, the open source **Verilog-to-Routing (VTR)** project was originally introduced and has been continuously evolving over the past 20+ years to enable FPGA architecture and CAD research via a highly flexible data-driven tool flow [27–29]. VTR takes as an input a human-readable description of a proposed FPGA architecture and produces physical (i.e., synthesized, placed, and routed) implementations of input designs on the described architecture with their corresponding estimates of the timing, area, and power consumption. This article presents the new VTR features and enhancements included in its latest release, *VTR 9*, to improve all three aspects of the tool flow: scalability, QoR, and modeling flexibility. Although some of these enhancements were previously introduced as independent contributions [30–36] and others are new additions to the flow, the main goal of this article is to describe how all these different efforts from tens of developers and multiple research institutes fit together in the broader picture of the VTR project, and provide a new baseline release for future research studies to compare to. The highlights of new VTR 9 features that we cover in this article are:

— New architecture modeling capabilities to capture recent commercial features such as embedded hard **Networks-on-Chip (NoCs)** and future architecture possibilities such as 3D-stacked FPGA fabrics.
— Architecture approximations of Intel's Stratix 10 and AMD series-7 FPGA fabrics to provide recent-generation baselines for researchers to build upon and enable approximate comparisons between VTR and commercial CAD tools on similar architectures.
— A new frontend flow, *Parmys*, which adds architecture awareness to Yosys [37] and improves automatic inference of hard blocks.
— Substantial enhancements to the VTR backend to optimize the overall generality, QoR, and runtime of the packing, placement, and routing stages.
— Multiple new benchmark suites to drive architecture and CAD studies using VTR such as the Koios suite for DL-specific designs, new Titan benchmarks targeting Stratix 10 commercial devices, and NoC-based benchmarks.

This article is organized as follows. First, Section 2 reviews related work, and Section 3 gives a high-level overview of the VTR flow. Then, Section 4 details the enhancements to VTR's architecture modeling capabilities, Section 5 introduces VTR-compatible benchmark suites, and Section 6 describes two new architecture captures of modern commercial FPGAs. Sections 7 and 8 summarize the enhancements to the VTR logic synthesis and physical implementation flows, respectively, while Section 9 covers developer-focused utilities and software engineering improvements. Section 10 provides a detailed evaluation of the VTR 9 CAD flow, offering an in-depth analysis of its performance and capabilities. Finally, conclusions and future work are presented in Sections 11 and 12, respectively.

2 Related Work

FPGA users implement application designs and generate the programming bitstream files for their FPGA devices using vendor-supplied closed-source CAD tools such as Quartus from Intel/Altera and Vivado from AMD/Xilinx. However, to enable researchers to evaluate new CAD algorithms, some FPGA vendors have defined and documented interfaces that facilitate the integration of research prototypes of certain stages of the design flow into the commercial tool flows. The now-legacy AMD/Xilinx Integrated Software Environment flow allowed users to access intermediate implementation results through the **Xilinx Design Language (XDL)** [38]. Torc [39] and RapidSmith II [40] leveraged XDL to introduce an open source C++ infrastructure for FPGA CAD research. Building upon this paradigm, RapidWright [41] adopted similar techniques for reading/writing of partial implementation results from/to different stages of Xilinx's modern Vivado CAD flow, enabling selective customization of specific implementation steps within the overall design flow. Intel/Altera documented interfaces in the Quartus University Interface Program for devices up to the Stratix III generation to allow users to access architecture/timing information and read and write technology mapping, placement, and routing results. However, these interfaces are not documented for more recent devices [42].

Although these interfaces provide valuable support to the research community, they have two limitations. Firstly, as they interface a research prototype implementation of a specific stage of the flow with closed-source implementations of other stages, they cannot be used to evaluate CAD algorithms that require (even minor) modifications to other parts of the flow. Secondly, they exclusively target existing commercial devices and thus do not allow the exploration of new FPGA architectures. Some vendors have internal tools to architect their next-generation devices like the Intel/Altera FPGA modeling toolkit [43], but these tools are proprietary and inaccessible by external researchers.

Several open source FPGA CAD tools have been developed to bridge these gaps. Independence [44] is a place and route tool developed to enable evaluation of new FPGA architectures; however, it's extremely long runtime limits its use to only small benchmarks. `Nextpnr` is a placement and routing tool that can target the Lattice iCE40, ECP5, and Nexus families; in combination with Yosys for logic synthesis, it enables an open source design implementation flow for these devices [45].

VTR [29] is an open source FPGA tool flow that has been widely used to explore different FPGA architectures and prototype new CAD algorithms; in recent years it has also been used as the design implementation/programming flow for both commercial and research FPGA devices. VTR's ability to model and target a wide variety of FPGA architectures makes it a common choice for exploring and evaluating new FPGA architectural ideas. For example, Eldafrawy et al. [25] use VTR to evaluate several LB architectural enhancements that increase the density of low-precision multiply-accumulate operations in the programmable fabric. Arora et al. [23] introduce new hard blocks for the tensor computations common in DL applications and use VTR to evaluate their effect on the resource utilization, routed WL, and maximum operating frequency of DL and non-DL benchmarks. Arora et al. [46] also use VTR to evaluate adding in-memory compute capabilities to FPGA BRAMs. Numerous works use VTR to explore programmable routing architecture enhancements, including direct inter-block connections [47], highly routable switch patterns [48, 49], diagonal wires [50], and capacitance-minimizing switch patterns [51].

Since VTR provides a complete open source FPGA CAD flow, it is also a popular framework in which new algorithms can be integrated and evaluated. For example, MULTIPART [52] proposes a

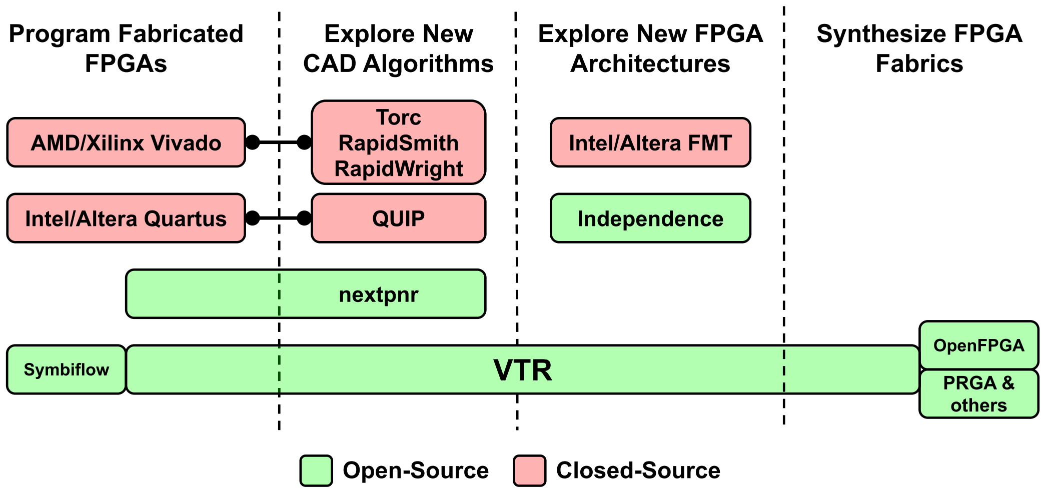


Fig. 1. Summary of the FPGA CAD tool features.

packing technique that combines seed-based and partitioning-based approaches. Elgammal et al. [53] introduce a reclustering API that allows changing the packing decisions in later stages of the flow. The authors of [54] apply machine learning to predict FPGA routing success, using a mix of binary classifiers and linear regressors. Chen et al. [55] create hooks between VTR and the OpenAI Gym framework to enable fine-grained interaction between CAD algorithms and **Reinforcement Learning (RL)** libraries. Other works use VTR to evaluate security threats and mitigations in open source CAD toolchains [56, 57].

VTR is also used as the implementation CAD flow for fabricated FPGA architectures. For example, the SymbiFlow project [58] compiles designs to configuration bitstreams for Xilinx's 7-series commercial FPGAs using Yosys [37] for logic synthesis and VTR for physical implementation (packing, placement, and routing). The QuickLogic Open Reconfigurable Computing CAD flow similarly uses Yosys for logic synthesis and VTR for physical implementation targeting their FPGAs. VTR is a part of the OpenFPGA [59] framework which automates the design, verification, and layout generation of user-specified FPGA architectures. OpenFPGA is now the main software tool chain used by multiple startups such as RapidSilicon and RapidFlex; the VTR architecture description drives the OpenFPGA layout generator, and the VTR CAD flow is used to implement designs in the fabricated FPGAs. VTR is also used in several research flows that synthesize custom standard-cell-based FPGA fabrics [60–64]. As summarized in Figure 1, VTR is the only open source flow that targets all these different FPGA CAD flow use cases.

Given the diverse applications of VTR and the large number of studies and products based on it, enhancements to its flexibility, result quality, and software engineering are all important.

3 Overall Flow

Figure 2 illustrates the overall flow of VTR 9, with the components upgraded or newly introduced in this release highlighted in red. The flow is segmented into two main stages: logic synthesis and physical implementation. The logic synthesis stage converts a digital circuit described in a HDL into a *technology-mapped* netlist composed of the basic resources (primitives) available in the target FPGA such as **Lookup Tables (LUTs)**, **Flip-Flops (FFs)**, and one-bit wide RAM slices. VTR 9 enhances the two logic synthesis flows (Odin/ABC and Titan) from prior releases and introduces a third flow (Parmys). The first flow, which was the default for VTR 8, uses Odin II

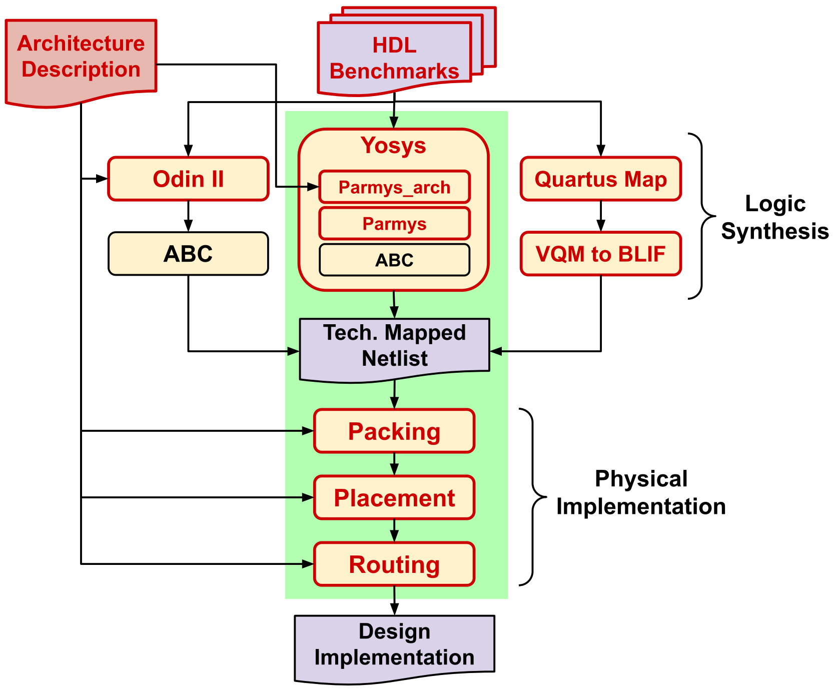


Fig. 2. Overview of the VTR 9 CAD flow. The components highlighted in red are updated in the new VTR release compared to prior versions and the default flow is highlighted in green.

for synthesis and ABC for technology mapping; in VTR 9, Odin II is enhanced to provide richer Verilog language support. Alternatively, the Titan flow uses the Intel/Altera Quartus frontend for synthesis and the `VQM2BLIF` utility to convert Quartus synthesis output to the BLIF format which can be consumed by the rest of the VTR flow. In previous VTR versions, the Titan flow only supported Intel's Stratix IV family, which is now only supported in older Quartus versions. In VTR 9, we provide an architecture capture of Stratix 10 (Section 6.1) and upgrade the `VQM2BLIF` utility to support Stratix 10 FPGAs and the latest Quartus Prime Pro versions. Additionally, we introduce the new Parmys logic synthesis flow (Section 7) that integrates Yosys with VTR and extends it by adding two new passes, `Parmys_arch` and `Parmys`, to combine Odin's architecture awareness with the richer language support of Yosys. This is the default logic synthesis flow of VTR 9.

The physical implementation stage in VTR 9 consists of three major components. Packing chooses how the primitive blocks in the technology-mapped netlist produced by logic synthesis are grouped into coarser-granularity blocks (i.e., *clusters*) to simplify the placement and routing problems. For example, LUTs and FFs are grouped into LBs while RAM slices are grouped into **Block RAMs (BRAMs)**. Placement then chooses the locations of clustered primitives on the grid of available *tiles* in the target FPGA such that the placement is legal (i.e., no two primitives are mapped to the same location and all primitives are mapped to the correct tile types) and the estimated WL and delay of the programmable interconnect used to implement all the inter-block connections are minimized. Finally, the routing stage chooses the exact programmable routing wires and switches for the connections between placed clustered primitives to minimize the overall WL and CPD.

This release provides major updates in each of these physical implementation flow steps. VTR 9's placement engine reduces runtime by combining a higher-quality initial placement (Section 8.1), several intelligent placement perturbations (Section 8.2), and a RL agent for adaptive and smart perturbation selection throughout the **Simulated Annealing (SA)** process (Section 8.3). The placement engine now optimizes not only traditional programmable fabric quality metrics but also NoC usage (Section 8.4). VTR 9 also allows users to define placement floorplanning constraints (Section 8.5), enabling both user-guided optimization and experimentation with divide-and-conquer physical implementation flows. In the routing step, VTR 9 enhances its state-of-the-art **Adaptive Incremental Router (AIR)** router [65] by introducing two major features: parallel routing and flat routing. Firstly, it can route the entire path between primitive pins in a single step (Section 8.7) to improve the QoR compared to the separate inter- and intra-cluster routing steps in the prior VTR versions. Secondly, it can now route multiple nets or portions of nets in parallel to reduce CPU time (Section 8.8). In addition, VTR 9 also extends the XML FPGA architecture description format to enable the modeling of modern commercial programmable routing fabrics (Sections 6.1 and 6.2), architectures with hardened packet-switched NoCs (Section 4.1), and 3D-stacked reconfigurable devices (Section 4.2).

Collectively, the VTR 9 enhancements allow more faithful captures of commercial FPGA architectures, enable exploration of new architecture features along with their CAD support, reduce the flow's runtime, and improve its QoR.

4 Architecture Modeling

In this VTR release, we extend VTR's XML architecture description language to enable exploration of new FPGA architectural features and their CAD support. For background on key FPGA architecture components and how they are modeled in VTR we refer interested readers to [17] and [66], respectively. The following subsections describe extensions which add support for hard packet-switched NoCs, 3D-stacked reconfigurable fabrics, different per-direction compositions of routing wire types, and complex wire shapes (e.g., L-shaped interconnect).

4.1 Hard NoCs

As FPGA devices continue to grow in capacity and complexity, it is becoming more challenging to close timing on the large number of modules communicating with each other and with various high-speed external interfaces (e.g., Ethernet, PCIe, high-bandwidth memories). Typically, an FPGA designer needs to implement and optimize system-wide communication busses, which requires several time-consuming design iterations and consumes a significant amount of the FPGA's programmable routing and registers. To face these challenges, recent commercial FPGAs from Xilinx [67], Achronix [68], and Intel [69] have introduced hard packet-switched NoCs embedded in their programmable fabric. These NoCs provide efficient system-level communication and facilitate timing closure. They can also enable faster FPGA implementation flows, in which different modules can be compiled in parallel and then stitched together using the system-wide NoC.

FPGA designs implemented on devices with embedded hard packet-switched NoCs contain modules connected to each other using the traditional programmable routing and others communicating by exchanging packets over the NoC, as illustrated in Figure 3. The circuit netlist contains not only the conventional primitives (such as LBs, RAM blocks) but also *logical routers* connected to the design modules that are sending and receiving packets over the NoC. During placement, the CAD tools are responsible for mapping these logical routers to *physical routers* at specific locations in the FPGA fabric to co-optimize conventional circuit metrics (e.g., CPD and WL) and NoC metrics (e.g., latency and aggregate bandwidth).

```xml
<!-- Description of a 3x3 mesh NoC-->
<noc link_bandwidth="128e9" router_latency="1e-9" link_latency="1e-9" noc_router_tile_name=
  "noc_router_adapter">
    <topology>
        <router id="0" positionx="0" positiony="0" connections="1 3"/>
        <router id="1" positionx="5" positiony="0" connections="0 2 4"/>
        <router id="2" positionx="10" positiony="0" connections="1 5"/>
        <router id="3" positionx="0" positiony="5" connections="0 4 6"/>
        <router id="4" positionx="5" positiony="5" connections="1 3 5 7"/>
        <router id="5" positionx="10" positiony="5" connections="2 4 8"/>
        <router id="6" positionx="0" positiony="10" connections="3 7"/>
        <router id="7" positionx="5" positiony="10" connections="6 8 4"/>
        <router id="8" positionx="10" positiony="10" connections="7 5"/>
    </topology>
</noc>
```

Listing 1. A snippet from a VTR architecture description file specifying a simple 3 × 3 mesh NoC.

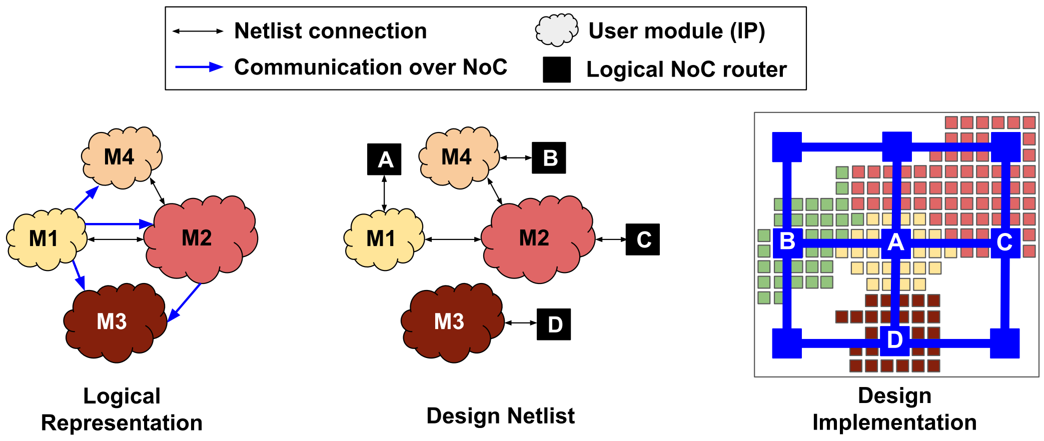


Fig. 3. Overview of the NoC placement optimization problem.

4.1.1 Flexible NoC Description. To enable architecture exploration of NoC-enhanced FPGAs in VTR 9, we extend the XML-based architecture description language to allow users to describe FPGA fabrics with embedded NoCs that have arbitrary topologies and specifications. Listing 1 shows an example architecture description file snippet that defines a simple 3 × 3 mesh NoC. The new `<noc>` tag can be used to specify general NoC specifications such as link bandwidth (in bits/s) and link/router latencies (in seconds). Then, the user can describe any custom NoC topology using the `<topology>` tag by listing all the NoC routers, their locations within the FPGA grid, and the IDs of the other routers they are connected to. This format enables specification of arbitrary topologies; for example, Figure 4 from the VTR GUI depicts an example NoC topology that resembles a simplified version of the AMD/Xilinx Versal NoC [67].

4.1.2 Traffic Flows Input File. For the VTR placement engine to also optimize NoC-related metrics such as latency and aggregate bandwidth, it requires a new design component: a description of the

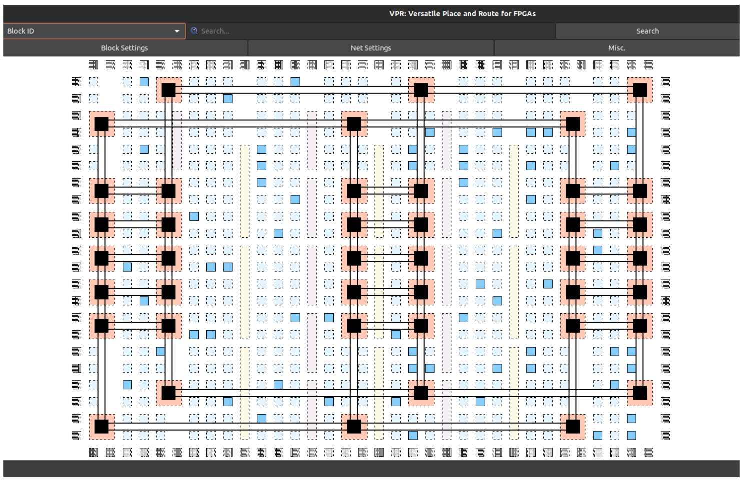


Fig. 4. VTR GUI showing an FPGA grid that contains a custom AMD-Versal-like NoC topology.

```
<traffic_flows>
    <single_flow src="m0" dst="m1" bandwidth="23e9" latency_cons="3e-9"/>
    <single_flow src="m0" dst="m2" bandwidth="50e8"/>
    <single_flow src="ddr" dst="m0" bandwidth="128e9" priority="3"/>
    <single_flow src="pcie" dst="m2" bandwidth="48e9" latency_cons="5e-9" priority="2"/>
</traffic_flows>
```

Listing 2. An example description of application design traffic flows.

application traffic expected to flow between different NoC endpoints. Thus, we add a new XML user input file to VTR that lists the source and destination router IDs and the bandwidth requirement for each of the NoC traffic flows as shown in Listing 2. Users can also optionally specify a priority and/or a latency constraint for a flow. A higher priority value for a specific traffic flow corresponds to a higher weight in the placement optimization cost function.

Section 8.4 details how VTR's placement optimization engine has been enhanced to leverage this architecture and design information to optimize NoC usage.

4.2 3D-Stacked Architectures

Recent advancements in chip packaging (e.g., Intel's Foveros [70] and TSMC's SoIC [71]) have enabled the 3D stacking of active dice on top of each other. This new technology has been employed in some GPU systems such as Intel's Ponte Vecchio and AMD's MI300 and some CPU systems such as Ryzen processors that contain a 3D V-Cache [72]. Future FPGAs could benefit from homogeneous stacking of multiple FPGA fabrics to increase logic capacity, or heterogeneous stacking of different dice such as an FPGA fabric with an ASIC application-specific accelerator die. While FPGAs were early adopters of 2.5D interposer-based integration [73], FPGA vendors have not yet utilized 3D die stacking technology, except for memory stacking in high bandwidth memory. 3D stacking enables higher interconnect bandwidth and lower delay between dice than 2.5D integration, making it attractive for future reconfigurable systems, but architecture exploration tools are needed to study how to best exploit it. Prior research by Ababei [74] augmented an older version of VTR with partitioning and 3D-awareness so it could evaluate homogeneous stacks of identical fabric dice. However, it could not evaluate the more general case of heterogeneous stacks in which each die

```xml
<!--FPGA on top of NoC die without programmable routing-->
<layout name="3d_heterogeneous_fpga" height="4" width="4">
  <layer die="0" has_prog_routing="false">
    <fill type="noc_die0">
  </layer>
  <layer die="1">
    <fill type="fabric_die1">
  </layer>
</layout>
<!--NoC router on layer 0 connects to layer 1-->
<tile>
  <sub_tile name="noc">
    <pinlocations pattern="custom">
      <loc layer_offset="1">noc.tdata[128:0]</loc>
    </pinlocations>
  </subtile>
</tile>
```

Listing 3. Sample XML representation of a 3D FPGA configuration. The lower die contains NoCs, and the upper die hosts the FPGA fabric.

can have different resource types and distributions. This earlier work was also not incorporated into the VTR master branch; it is neither available online nor compatible with recent VTR versions.

VTR 9 fills this gap by extending the architecture description file to define 3D architectures and enhancing the placement and routing engines to exploit the flexibility provided by 3D devices [31]. We extend the VTR architecture description language so that it now can describe a stack of dice in which each die has its own layout (i.e., grid) of available resources; this allows modeling of an arbitrary homogeneous or heterogeneous stack of dice. The architecture file also allows the user to define arbitrary connectivity between the different layers. For each block type, some or all of its inputs and outputs can have programmable switches connecting them to routing wires on other dice. In addition, custom **Switch Blocks (SBs)** can be specified to create arbitrary 3D switch patterns in which some programmable routing switches allow inter-die crossings between routing wires. For example, Listing 3 illustrates a portion of an XML file that describes a 3D architecture with two dice using the `<layer>` tags. In this example, an FPGA fabric is stacked on top of a die that contains a NoC. The pins of NoC blocks located on the lower die (die#0) can connect to the programmable routing wires of the FPGA fabric on the upper die (die#1).

4.3 Heterogeneous Programmable Routing

In many commercial architectures both the channel width and the wire segment types in the horizontal and vertical directions are different for a variety of reasons, such as different metal layer usage or layout efficiency. For example, the horizontal channel width in Stratix 10 devices is nearly twice that of the vertical channels. Stratix 10 uses a redundant row repair scheme that requires extra (redundant) connections on vertical wires such that broken rows can be skipped without recompiling a design; this makes vertical wiring more expensive, motivating wider horizontal channels and narrower vertical ones. Another common reason for differing channel widths is that the ratio of rows to columns is often not 1:1 due to the layout aspect ratio of the FPGA tiles. In Stratix 10, for example, the LB is roughly twice as tall as it is wide. A physically square Stratix 10 FPGA would therefore have more columns than rows, which increases demand for horizontal routing and is most efficient with wider horizontal channels [75]. The tall and skinny Stratix 10 LB also makes horizontal wires physically shorter than vertical wires of the same logical (SBs spanned)

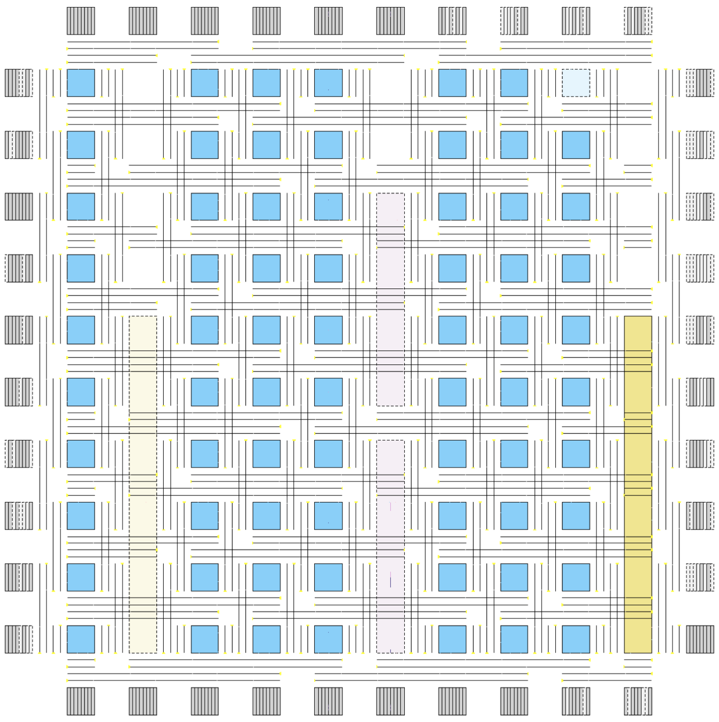


Fig. 5. VTR GUI showing an FPGA grid with different wire types in the vertical (length-2) and horizontal (length-4) directions as described in Listing 4.

length [43]. Therefore, horizontal channels consist of length 2, 4, 10, and 24 wires, while vertical channels consist of (logically shorter) length 2, 3, 4, and 16 wires.

In previous versions of VTR, only routing architectures with identical horizontal and vertical channel widths and types were supported. To address this shortcoming, we extend the VTR architecture description to allow users to describe different channel widths and wire segment types in the horizontal and vertical directions, as shown in Listing 4. The `<chan_width_distr>` tag in this XML snippet specifies that vertical channels are only 48% as wide as horizontal channels, and the `<segment>` tags define an x-directed length 4 wire and a y-directed length 2 wire using the new `axis` attribute. Figure 5 shows the resulting FPGA architecture in the VTR GUI.

To enable this more flexible architecture specification, the RR graph generator in VTR is extensively upgraded and the data-driven optimizers in VTR automatically adapt to it. The placement engine's WL cost divides the expected wiring in a region/direction by the average channel capacity in that area, biasing it to minimize wiring where capacity is most limited. The placer and the router profile paths through the RR graph to create fast timing and resource estimators before their main algorithms begin, enabling automatic adaptation to different wire types and directionality in the architecture.

4.4 Complex-Shape Routing Wires

Previous versions of VTR have supported the specification of a variety of routing segment lengths in the architecture XML file, but only traveling in the cardinal directions (north, east, south, and west), with no support for other shapes like bent (L-shaped) wires or T-shaped wires. However, academic research has shown timing and area benefits to more complex wire shapes [76], and

```
<device>
  <chan_width_distr>
    <x distr="uniform" peak="1.00"/>
    <y distr="uniform" peak="0.48"/> <!-- vertical channels only 48% as wide as horizontal -->
  </chan_width_distr>
</device>

<!-- This architecture sets routing wire Cmetal and Rmetal to 0, as all switches are buffered
     and each switch delay (not shown) includes the delay of the wire it drives. The x- and y-
     directed wires below have different lengths. -->
<segmentlist>
  <segment Cmetal="0" Rmetal="0" axis = "x" freq="2" length="4" name="H4" type="unidir">
    <mux name="0"/>
    <sb type="pattern">1 1 1 1 1</sb>
    <cb type="pattern">1 1 1 1</cb>
  </segment>
  <segment Cmetal="0" Rmetal="0" axis = "y" freq="2" length="2" name="V2" type="unidir">
    <mux name="0"/>
    <sb type="pattern">1 1 1</sb>
    <cb type="pattern">1 1</cb>
  </segment>
</segmentlist>
```

Listing 4. A snippet from a VTR architecture description file specifying different horizontal (X) and vertical (Y) channel widths and wire types.

many commercial FPGAs contain bent wire segments to create more complex routing patterns. For example, Xilinx 7-series FPGAs contain wires that travel one tile vertically and then turn and travel one tile horizontally (called NW2, NE2, SW2, SE2), as well as wires that travel two tiles horizontally and four tiles vertically (called NW6, NE6, SW6, SE6) [77]. Neither VTR 8's XML architecture file nor its **Routing Resource (RR)** graph generator supported the specification of these *bent/diagonal* wires. The VTR 8 router can leverage these wires if they exist in a custom (externally provided) RR graph, but the inability to specify and generate RR graphs with complex wire shapes within VTR restricted architecture exploration of FPGAs containing them.

In the VTR 9 architecture file, users can define custom SBs with switch types that join a set of horizontal and vertical wires using non-programmable (i.e., shorted) connections, while maintaining normal switch connections at other points (e.g., the ends) on these wires. This allows the formation of arbitrary rectilinear wire shapes such as those illustrated in Figure 6. These wires can be described in the architecture XML file as shown in Listing 5 for an NE3 (L-shaped) case.

5 Benchmark Suites

To evaluate the QoR of a proposed architecture or a new CAD algorithm, we need a suite of benchmark designs that are representative of contemporary FPGA use cases. To ensure CAD or architecture conclusions are applicable to a broad range of designs, it is important for these benchmarks to be diverse in circuit properties (e.g., routing demand, maximum logic depth, fanout connections), FPGA resource composition (e.g., memory, **Digital Signal Processing [DSP]**, registers), and application domain (e.g., wireless communications, DL, emulation). Large and complex benchmarks best represent current and future designs, but it is also beneficial to have some smaller designs that allow rapid iteration during the initial stages of architecture or CAD tool tuning. FPGA architectures with new hard blocks (e.g., NoC routers) also necessitate new benchmarks that use these resources to enable quantitative evaluation. The VTR [27, 29] and Titan23 [78] suites are

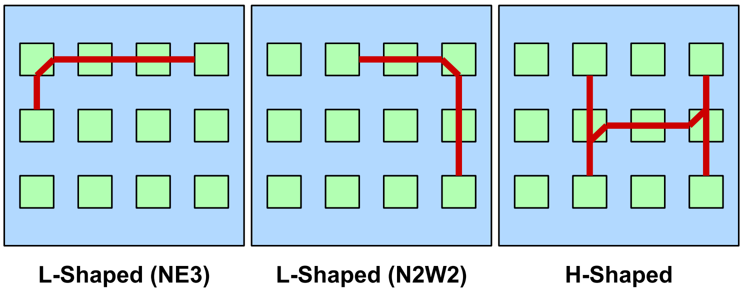


Fig. 6. Example variations of complex-shape wires in VTR 9. Green squares represent FPGA SBs and red lines are complex-shape wires formed by shorting multiple cardinal wire segments together. The rest of the programmable routing fabric is not shown for clarity. L-shaped wires short two cardinal wires together and H-shaped wires short 3; arbitrary wire shapes can be created by shorting any number of cardinal wires together.

```
<segmentlist>
  <segment axis="y" name="NE3_vert" freq="0.04" length="1" type="unidir" Rmetal="101"
          Cmetal="22.5e-15"> ... </segment>
  <segment axis="x" name="NE3_horiz" freq="0.04" length="3" type="unidir" Rmetal="101"
          Cmetal="22.5e-15"> ... </segment>
</segmentlist>

<switchblocklist>
  <switchblock name="NE3_diagonal_short" type="unidir">
    <switchblock_location type="EVERYWHERE" />
    <!-- Create a short from an NE3_vert (track t bottom) wire to an NE3_horiz (track t right)
         wire to make an L-shaped wire -->
    <switchfuncs>
      <func type="br" formula="t"/>
    </switchfuncs>
    <wireconn num_conns="min(from,to)" from_type="NE3_vert" from_switchpoint="0"
      to_type="NE3_horiz" to_switchpoint="0" switch_override="electrical_short"/>
  </switchblock>

  <switchblock name="NE3_normal" type="unidir">
    <switchblock_location type="EVERYWHERE"/>
    <!-- Define switchblock functions for normal interconnect at ends of diagonal wire -->
  </switchblock>
</switchblocklist>
```

Listing 5. A snippet from a VTR architecture description file specifying an L-shaped (NE3) routing wire.

the most commonly used benchmark suites in the FPGA literature and are supported by VTR 9. To keep pace with the evolution in FPGA applications, VTR 9 includes an additional Titan-flow suite with new designs (*Titanium25*), a DL focused suite (*Koios*), and a suite of designs that include NoC communication (*Hermes*). All these benchmarks (summarized in Table 1) have been made compatible with the VTR CAD flow and are included in its tests to ensure they remain compatible.

5.1 Titanium25: Large Stratix Benchmarks

The Titan23 benchmark suite has been a part of VTR releases since VTR 7. In VTR 9, we augment the Titan23 suite with two main improvements. First, we incorporate 25 additional benchmarks

Table 1. Summary of the New Benchmark Suites in VTR 9

Suite	# Circuits	Domain	# Primitives
Titanium25 (S-IV & S10 Arch)	25	Misc.	23K–7.6M
Koios (Arch Agnostic)	40	DL	
Application Designs	32		12K–1.6M
Synthetic Benchmarks	8		148K–440K
Hermes (Arch w/ NoCs)	35	NoC Comm.	
Synthetic (Known Optimal Sol.)	27		5K–201K (4–64 R)
Peripheral Interfaces (I/O Heavy)	4		210K (64 R)
MLP Acceleration (DL)	4		316K–579K (9–16 R)

Table 2. Characteristics of the Titanium25 Benchmark Circuits

Name	# Primitives	LUTs	Reg.	DSP	Mem. Bits	Application
`mem_tester_max`	7,605,183	3,442,316	4,002,852	0	163,811,328	Mem. parametric failure testing
`rocket31`	1,448,187	933,803	481,672	341	8,506,654	31-core RISC-V Rocket chip
`ASU_LRN`	955,146	346,955	596,579	3,036	4,165,888	AlexNet [79] accelerator
`ChainNN_LRN`	937,695	190,550	428,996	2,437	16,874,048	AlexNet accelerator
`ChainNN_ELT`	937,300	199,771	419,571	2,310	16,777,216	ResNet-50 accelerator
`ChainNN_BSC`	905,098	185,247	401,893	2,310	16,777,216	VGG-16 accelerator
`rocket17`	801,897	514,910	268,114	187	4,643,612	17-core RISC-V Rocket chip
`ASU_ELT`	767,837	262,976	493,697	3,036	3,965,184	ResNet-50 [80] accelerator
`ASU_BSC`	734,883	245,975	477,744	3,036	3,965,184	VGG-16 [81] accelerator
`tdfir`	706,338	309,835	394,083	512	211,680	DSP
`pricing`	668,537	290,371	350,454	774	3,769,552	Option pricing algorithm
`mem_tester`	621,351	283,304	325,227	0	163,811,328	Mem. parametric failure testing
`mandelbrot`	579,813	214,447	348,072	250	1,181,672	Fractal rendering
`channelizer`	462,003	179,655	265,878	196	6,530,084	DSP
`fft1d_offchip`	440,661	165,076	258,706	166	5,165,932	DSP spectral analysis
`DLA_LRN`	414,250	151,150	255,660	400	14,108,160	AlexNet accelerator
`matrix_mult`	392,682	157,112	219,894	264	3,437,992	Matrix multiplication
`fft1d`	389,350	150,918	227,580	120	4,612,592	DSP spectral analysis
`fft2d`	354,506	132,760	209,273	96	9,978,604	DSP spectral analysis
`neko`	304,581	184,628	113,215	178	6,717,440	GPU simulation
`DLA_ELT`	296,292	93,313	195,763	304	14,011,392	ResNet-50 accelerator
`DLA_BSC`	285,347	87,498	190,633	304	14,011,392	VGG-16 accelerator
`jpeg_deco`	209,313	79,508	110,831	145	3,967,482	Image processing
`nyuzi`	90,857	47,987	37,459	96	1,529,368	GPGPU processor
`sobel`	23,224	8,575	12,717	0	131,464	Image processing

(collectively, Titanium25) with an average primitive count of 850k; these new designs include recent applications such as DL accelerators which are not represented in the original Titan23 suite. Table 2 lists key characteristics of the Titanium25 benchmarks. Both Titan23 and Titanium25 include Intel/Altera-specific IPs, and hence use Quartus synthesis plus VTR placement and routing (i.e., the Titan flow) [78]. Secondly, we implement the Titanium25 designs and update the 23 original Titan designs so that both are now compatible not only with Stratix IV devices (as in VTR 8) but also with Stratix 10 devices. The Stratix 10 versions allow compilation in more recent commercial tools (e.g., Quartus Prime Pro 23).

5.2 Koios: DL Benchmarks

With the continuous growth in DL-driven applications both in datacenters and edge devices, DL acceleration has become a major FPGA use case [13, 82] but neither the VTR nor the Titan23 benchmark suites include designs from this domain. The Koios suite addresses this gap by introducing 40 DL benchmark circuits that capture a wide variety of sizes, circuit properties, target neural networks, and numerical precisions. Some of these circuits are original implementations, some are re-creations of published designs from industry and academia, and others are collected from open source implementations. The benchmark circuits make heavy use of DSP and BRAM blocks and their sizes range from 12K to 1.6M netlist primitives. Unlike the Titan23 and Titanium25 benchmarks, the Koios circuits are compatible with the VTR 9 front-end (Parmys) and do not need to use Quartus for synthesis. Thus, they are not limited to only the Stratix IV and Stratix 10 architecture captures and can be flexibly used with arbitrary FPGA architecture definitions. For more details about the characteristics of these benchmarks and their use for FPGA architecture and CAD studies, we refer the reader to [35].

5.3 Hermes: NoC Benchmarks

As detailed in Section 4.1, many recent FPGA families include hardened NoCs within their fabric; however, none of the pre-existing benchmark suites include designs that leverage an NoC for communication. To enable quantitative evaluation of NoC-aware CAD and NoC-enhanced architectures, VTR 9 builds on the work in [30, 83] to create the Hermes benchmark suite. It consists of three categories of benchmarks, as summarized in Table 1. The first category consists of 27 synthetic benchmarks with different numbers of routers (listed in parenthesis under # primitives in Table 1) and different communication patterns between routers. Each design employs traffic generator modules (to send data to a NoC router) and traffic processor modules (to receive data from NoC routers). The traffic flows in each design follow one of three patterns: 1D chain, 2D nearest neighbor, or star communication. The optimal placement for each design, in terms of NoC metrics, can be manually determined, aiding in evaluating the NoC-aware placement engine. The second category has four peripheral interface benchmarks whose communication patterns are inspired by real-world applications. In each design, a subset of NoC routers are locked down to locations at the boundary of the FPGA device to mimic hardened memory controllers and high-speed interfaces, such as PCIe. Finally, Hermes includes four **Multi-Layer Perceptron (MLP)** inference designs in which large compute-intensive modules communicate using a hard NoC. These designs help assess NoC-aware CAD flows and FPGA architectures for larger, more realistic applications.

6 Architecture Capture of Commercial FPGAs

Modeling recent commercial FPGA architectures in VTR serves several purposes. First, it exercises the architecture modeling capabilities of VTR, driving enhancements to its generality and flexibility. Second, these architecture captures can be modified by researchers to evaluate a new feature in the context of a realistic and complete FPGA. Finally, these captures enable the use of the synthesis frontend of commercial tools (as in the Titan flow [78]), which in turn facilitates developing new benchmarks for evaluating VTR's QoR and approximate VTR to commercial QoR comparisons to identify areas for improvement. In VTR 9, we include architecture captures of two commercial FPGA families, one from Intel/Altera and another from AMD/Xilinx; the modeling of essential architecture features of these devices is discussed in the following subsections.

6.1 Intel/Altera Stratix 10 Architecture

Earlier versions of VTR included a capture of the Stratix IV (40 nm) FPGA family. However, Stratix IV is not supported by the recent versions of the Quartus Prime Pro tool suite from Intel/Altera and its routing architecture is simpler, with fewer types of routing wires, than more recent FPGAs. In addition to the Stratix IV architecture capture, VTR 9 includes a capture of the 14 nm Stratix 10 FPGA family [84] which contains a more diverse set of routing wires and is supported by the most recent CAD tool versions. The next sections discuss the various components of this architecture capture, which is contained in the `stratix10_arch.timing.xml` file in the VTR repository.

6.1.1 Logic Array Blocks (LABs). Each of the programmable LABs in Stratix 10 contains 10 **Adaptive Logic Modules (ALMs)**. Each ALM consists of a 6-input **Fracturable Lookup Table (FLUT)**, two full adders, and four output registers. The *mode* features in the VTR architecture description are used to capture the fact that an ALM can be used in several ways (one 6-LUT, two 5-LUTs, LUTs feeding adders, etc.). Each LAB includes 60 local interconnects called LAB lines, driven by general routing, neighboring blocks, and the LAB's own outputs. In our Stratix 10 capture, we assume a full crossbar between LAB lines and ALM inputs, which will result in a slight overestimation of routability compared to the 50% populated crossbars in the commercial device [33]. One-quarter of the LAB columns can also function as memory blocks (MLABs), with all their 10 ALMs collectively configured as a 64 × 10b or 32 × 20b memory unit. VTR 9 adds a new *equivalent tiles* feature that enables precisely capturing such dual-use blocks, which was not possible in earlier versions.

6.1.2 Hard Blocks. The Stratix 10 DSP blocks have **Input/Output (I/O)** registers, internal pipeline registers, multipliers, adders/subtractors, and an accumulator. These components can be configured to operate in many different ways; however, modeling all possible DSP configurations is cumbersome and results in a significantly more complex architecture description file despite many of the configurations being very similar. As a result, we capture only the key operating modes of the DSP block based on their function (e.g., multiply-accumulate, independent multiplier) and whether the inputs and outputs are registered or not. The Stratix 10 devices also contain a single type of BRAM with 20Kb capacity (M20Ks), for which we capture several configurations based on the operating mode (e.g., single-port, true dual-port) and whether the output ports are registered or not. Our capture of M20Ks does not include the less commonly used configurations such as the simple quad-port and dual-port ROM modes.

6.1.3 I/Os. The I/O banks are located in two dedicated columns in the Stratix 10 fabric. Each I/O bank consists of four I/O lanes, with each lane containing 12 **I/O Elements (IOEs)**. An IOE consists of various components such as I/O pads, I/O buffers, an output enable register, double data rate I/O circuitry, and a pseudo-differential buffer. Groups of I/Os share **Phase Locked Loop (PLL)** and on-chip termination controller blocks. While our benchmark circuits currently use only simple I/Os that do not leverage most of these blocks, the architecture capture includes them all to enable the use of future benchmarks with complex I/Os.

6.1.4 Device Layout. The architecture's layout follows the structure of the Intel Stratix 10 GX/SX series which mainly consists of FPGA core fabric (including LAB, MLAB, M20K, and DSP columns), I/O banks, transceivers, a **Secure Device Manager (SDM)**, and in some variants, a **Hard Processor System (HPS)**. In our capture, the transceivers, SDM and HPS are replaced with empty regions as they typically have a marginal impact on QoR and are not used in our benchmark suites. Figure 7 shows a snapshot of the device layout from the VTR GUI.

6.1.5 Routing Architecture. All the programmable routing wires in Stratix 10 are x- or y-directed, with different types of routing segments in each direction. The routing wire drivers also have optional registers (known as the hyperflex registers [84]) that enable deep pipelining application designs to improve timing. However, we do not include these interconnect registers in our architecture capture since the physical implementation algorithms in VTR cannot make use of them yet. The Stratix 10 routing architecture has four types of wires per direction. The different types of wires and their composition in the horizontal and vertical channels are listed in Table 3. Table 4

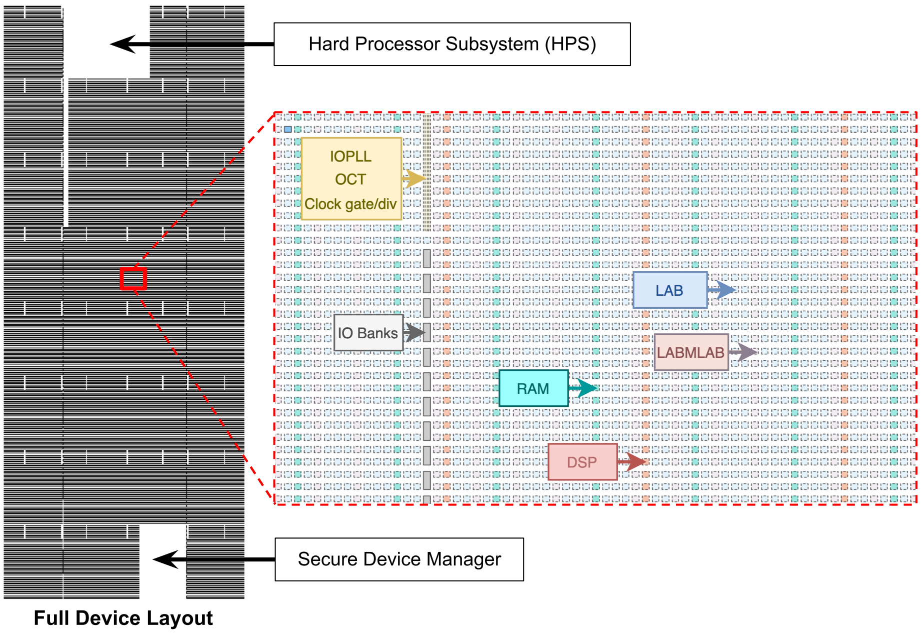


Fig. 7. Screenshot of our captured Stratix 10 architecture layout from the VTR GUI.

Table 3. Breakdown of Stratix 10 Routing Channels by Wire Type

Horizontal Wires			Vertical Wires		
Type	**Count**	**% of Channel**	**Type**	**Count**	**% of Channel**
H2	40	10%	V2	24	12%
H4	112	28%	V3	72	38%
H10	200	50%	V4	64	33%
H24	48	12%	V16	32	17%

Table 4. Organization of Driver Multiplexers for Different Wire Types in Stratix 10 Architecture Capture

		Distribution of Driver Mux Inputs			
Wire Type	**Driver Mux**	**H2, H4, H10**	**V2, V3, V4**	**H24, V16**	**Block OPins**
H2, H4, H10	12:1	2	4	1	5
V2, V3, V4	15:1	7	1	1	6
H24, V16	50:1	38	–	12	–
Block IPins	12:1	6 or 7	4	–	2 or 1

summarizes the size of the multiplexer that drives each type of wire or block input pin, along with the statistical distribution of how many of these multiplexer inputs come from each type of wire and from block output pins. Our architecture capture matches the distribution of wire types and switches driving each kind of wire in Stratix 10, but not the precise switch pattern of which exact wires drive each other, as that information is not public. Stratix 10 also includes nearest neighbor connections between adjacent blocks that bypass the general routing, and dedicated routing for carry chains and DSP cascade chains; both these features are modeled in our architecture capture.

6.1.6 Timing Models. To model the timing of different architecture components, we develop a number of microbenchmark circuits that are mapped by Quartus to different combinations of primitives operating in various modes. Then, we extract delays from the "Slow 900mV 100C" timing corner, which typically resulted in the worst-case delays. The local interconnect, LUT, and register delays are captured for the LABs, with different delay values for the paths from each LUT input to the output. For M20Ks, we extract the timing models for any combination of operating modes (e.g., ROM, single-port RAM, dual-port RAM) and output port configuration (e.g., combinational, registered). We also extract the timing information of different operating modes of the DSP block but to reduce modeling complexity, we average the delay values of similar modes under one mode in our architecture capture. Finally, since the metal wire parasitics of the Stratix 10 global routing is unknown to us, we extract the average combined delay for each wire type and its driver from the Quartus reports of a large set of benchmarks. Then, we use these values as the driver delays for the corresponding types of wires and set the wire delay itself to zero.

6.1.7 Validation against Quartus Results. Table 5 shows quality and runtime metrics of the Titan-New designs when implemented by VTR in this architecture capture, with the relative comparison to Quartus Prime Pro 23.4 targeting Stratix 10 in brackets. The mem_tester_max design is too large to fit in any Stratix 10 device and the rocket31 and neko designs were timed out in VTR after 2 days (during the routing phase) and were therefore excluded. Both VTR and Quartus Prime Pro target an unachievable design frequency to maximize design performance and use one CPU core on a 3.9 GHz Xeon Gold 6146 server with 768 GB of RAM. Quartus is run with default settings, while VTR was run with a higher placement effort (`--inner_num 2`) and a less directed routing search (`--astar_fac 1`) to align its effort level (which controls QoR vs. Runtime) more closely with that of Quartus. Since the Titan flow uses Quartus for synthesis, the primitive counts are the same for VTR and Quartus. Table 5 shows that the LB packing density is similar between Quartus and VTR, and the DSP packing is identical as each Stratix 10 DSP block contains only a single primitive, but VTR uses 14% more M20K BRAMs than Quartus. VTR uses somewhat more WL (1.06×) than Quartus but has 32% less runtime despite the higher-effort VTR placement and routing settings used in this comparison. The largest quality gap occurs for CPD, which is 28% higher in VTR than Quartus on average. There are many possible reasons for this gap, but significant contributors are likely that the hand-tuned routing switch patterns in Quartus are more optimized than the automatically generated VTR switch patterns, that VTR is not modeling or using the Stratix 10 interconnect registers, and Quartus includes retiming algorithms while VTR does not.

6.2 AMD/Xilinx 7-Series Architecture

While most architecture descriptions shipped with previous VTR versions have been modeled after Intel/Altera FPGA families, VTR 9 introduces several new architecture generation features that allow modeling of AMD/Xilinx FPGAs such as support for different wire types/distributions in the horizontal and vertical channels (Section 4.3) and L-shaped wires (Section 4.4). We leverage these features to develop an approximate architecture model for AMD/Xilinx 7-series FPGAs which can be found in the `7series_BRAM_DSP_carry.xml` file in the VTR repository. The following subsections outline the various components of this architecture capture and their current limitations.

6.2.1 Configurable Logic Blocks (CLBs). In our architecture capture, each CLB consists of two slices, each of which has four **Basic Logic Elements (BLEs)**. A BLE is a 6-input FLUT (that operates as a 6-LUT or two 5-LUTs) combined with two FFs and one bit of a carry chain [17, 85]. The 7-series CLB contains two disjoint carry chains (one for each slice), enabling more flexible logic packing; the architecture capture models this pattern of carry chains and their dedicated interconnect cascading across the CLBs in a column. Similarly to the Virtex-6 architecture [86], each

Table 5. Implementation Results of the Titanium25 Benchmarks on Our Stratix 10 Architecture Capture

Benchmark	Resource Utilization						CPD (ns)		WL		Run Time (s)	
	LABs		DSPs		BRAMs							
ASU_LRN	41,079	(1.36×)	3,036	(1.00×)	372	(1.00×)	10.69	(2.79×)	10,034,904	(1.08×)	28,903	(0.87×)
ChainNN_LRN_LG	47,087	(1.02×)	2,438	(1.00×)	2,395	(1.00×)	8.52	(1.14×)	6,595,885	(0.96×)	20,393	(0.51×)
ChainNN_ELT_LG	46,298	(0.99×)	2,310	(1.00×)	2,392	(1.00×)	9.31	(1.68×)	6,745,044	(1.00×)	20,723	(0.59×)
ChainNN_BSC_LG	45,528	(0.99×)	2,310	(1.00×)	2,392	(1.00×)	6.51	(1.31×)	6,830,877	(0.98×)	19,849	(0.53×)
rocket17	47,005	(0.78×)	187	(1.00×)	1,233	(3.30×)	14.58	(1.13×)	10,347,803	(0.97×)	48,463	(1.01×)
ASU_ELT	33,143	(1.45×)	3,036	(1.00×)	344	(1.00×)	7.20	(2.04×)	7,598,316	(0.94×)	22,931	(0.88×)
ASU_BSC	31,837	(1.44×)	3,036	(1.00×)	344	(1.00×)	8.81	(2.43×)	7,154,581	(0.86×)	18,136	(0.72×)
tdfir	26,529	(0.95×)	512	(1.00×)	56	(1.12×)	5.73	(0.95×)	4,106,740	(1.31×)	22,468	(1.15×)
pricing	25,379	(0.95×)	774	(1.00×)	574	(1.02×)	5.56	(1.50×)	3,924,362	(1.27×)	21,101	(1.11×)
mem_tester	25,532	(0.92×)	0	–	1,601	(1.00×)	1.80	(0.95×)	1,608,865	(1.18×)	11,786	(0.94×)
mandelbrot	22,459	(0.99×)	250	(1.00×)	141	(1.01×)	5.61	(1.20×)	3,054,543	(1.24×)	17,371	(1.09×)
channelizer	16,753	(0.95×)	196	(1.00×)	588	(1.03×)	5.31	(0.93×)	2,107,209	(1.16×)	10,427	(0.79×)
fft1d_offchip	16,683	(0.99×)	166	(1.00×)	466	(1.03×)	5.94	(1.11×)	2,211,044	(1.20×)	11,085	(0.89×)
DLA_LRN	17,606	(1.03×)	448	(1.00×)	824	(1.00×)	5.94	(1.91×)	2,347,767	(1.00×)	9,565	(0.67×)
matrix_mult	14,682	(0.96×)	264	(1.00×)	1,527	(2.50×)	4.91	(0.97×)	2,707,265	(1.35×)	9,561	(0.78×)
fft1d	14,268	(0.95×)	120	(1.00×)	367	(1.04×)	4.92	(1.04×)	1,701,507	(1.13×)	7,732	(0.69×)
fft2d	12,944	(0.91×)	96	(1.00×)	714	(1.12×)	4.92	(0.98×)	1,971,169	(1.19×)	7,275	(0.66×)
DLA_ELT	12,523	(1.15×)	352	(1.00×)	816	(1.00×)	5.75	(1.15×)	1,721,404	(1.06×)	5,893	(0.57×)
DLA_BSC	12,183	(1.14×)	352	(1.00×)	816	(1.00×)	5.49	(1.04×)	1,652,062	(1.04×)	5,855	(0.59×)
jpeg_deco	8,635	(0.95×)	145	(1.00×)	686	(1.08×)	6.69	(1.31×)	1,391,682	(0.96×)	5,051	(0.62×)
nyuzi	4,468	(0.80×)	96	(1.00×)	280	(1.39×)	8.51	(1.35×)	1,316,746	(1.07×)	1,949	(0.32×)
sobel	834	(0.82×)	0	–	53	(1.04×)	4.79	(0.89×)	107,052	(0.61×)	457	(0.15×)
Geomean	1.01×		1.00×		1.14×		1.28×		1.06×		0.68×	

The ratios between brackets compare to Quartus Prime Pro results targeting a Stratix 10 device.
`mem_tester_max` is too large for any Stratix 10 device; the VTR runs of `rocket31` and `neko` were timed out after 2 days.

7-series BLE output can connect back to six BLE inputs in the same CLB using the local interconnect. Half of the LBs in a 7-series FPGA can also implement distributed RAM; these are called `SLICEM` blocks whereas the logic-only version is called a `SLICEL`. Currently, our architecture capture does not support distributed RAM and models all CLBs as `SLICEL` blocks. All the AMD/Xilinx 7-series CLBs also contains `MUX7` and `MUX8` multiplexer elements that allow 6-LUT outputs in the same CLB to be combined to form larger LUTs. These multiplexers are not yet included in our architecture capture; they are less important than the LUTs and have been removed in the more recent Versal architecture [87].

6.2.2 *Hard Blocks.* We capture the most important operating modes of the AMD/Xilinx 7-series BRAMs (`RAMB36E1` and `RAMB18E1` primitives). In their true dual-port mode, the `RAMB36E1` primitives can be configured as 32K × 1b, 16K × 2b, 8K × 4b, 4K × 9b, 2K × 18b, or 1K × 36b memories. An additional 512 × 72b configuration is available in the simple dual-port mode. Each BRAM tile can also implement two `RAMB18E1` primitives which have configurations with the same widths but half the depths of those of the `RAMB36E1` primitive. Our architecture capture does not include the dedicated cascades between BRAMs in the same column (which enable the implementation of deeper memories without using any extra programmable logic or routing) or the support for mismatched read and write word widths. We also model a simplified version of the 7-series DSPs that captures the 25 × 18 multiplication mode without support for the pre-addition or ALU functionalities.

6.2.3 *Routing Architecture.* To capture the details of the 7-series routing architecture, we combine information from the NetCracker project [77], the Project X-ray database [88], and our own custom `tcl` scripts that query information from Vivado. The distribution of straight (prefixed with an “L”) and multi-cardinal/diagonal (prefixed with an “MC”) wire types is summarized in Table 6. The

Table 6. Breakdown of 7-Series Routing Channels by Wire Type

Horizontal Wires (W_h = 124)			**Vertical Wires (W_v = 190)**		
Type	**Count**	**% of Channel**	**Type**	**Count**	**% of Channel**
L1	14	11.3%	L1	14	7.4%
L2	16	12.9%	L2	16	8.4%
L4	32	25.8%	L4	0	0%
L6	0	0%	L6	48	25.2%
L12	12	9.7%	L12	12	6.3%
L18	0	0%	L18	18	9.5%
MC1	18	14.5%	MC1	18	9.5%
MC2	32	25.8%	MC2	0	0%
MC4	0	0%	MC4	64	33.7%

The absolute wire counts reflect those of the actual 7-series architecture; however, our VTR capture enables targeting architectures with the same wire type composition but different total channel widths.

number following the “L” or “MC” prefix indicates the length the wire spans in a cardinal direction. For example, each length six diagonal wire consists of a vertical MC4 and a horizontal MC2 hard-wired together with a non-programmable switch that represents a metal short. Similarly, two MC1 wires (one vertical and one horizontal) are shorted to create a length-2 diagonal connection.

The coordinate system and tiling structure of the 7-series architecture is different from those conventionally used by VTR [51]. The 7-series pairs LBs with **Interconnect Tiles (INT)** in a single repeatable structure. INTs contain the programmable switches used for connecting routing wires to block I/O pins as well as those connecting different routing wires. In contrast, VTR uses the notion of SBs and connection blocks and local routing is usually modeled within the LB itself. Figure 8 illustrates key features of the 7-series routing architecture and how they are modeled in VTR. For example, a vertical L1 7-series wire is driven by one INT and reaches one INT above/below, enabling connections to two different routing channels and two different CLBs. To model equivalent reach, a VTR wire needs to be of length 2. Thus, in our 7-series VTR architecture description file, each cardinal direction wire is modeled as one unit longer than what its 7-series name in Table 6 would imply. Some of the cardinal wires within the 7-series have access to two adjacent SBs at their ends; these extra connections are termed *stubs* in [77]. One-eighth of L1 wires and one-fourth of the other straight (L) wires have stubs, which we model by shorting a perpendicular L1 to the end of the wire. Cardinal wires with stubs are similar to the MC wires but MC wires connect to a single SB at their end. The L12 and L18 wires in the commercial 7-series architecture are bidirectional, which means they can be driven from either end of the wire segment. While VTR supports both directional and bidirectional FPGA routing architectures, the architecture generator does not currently support mixing these two types. Hence our architecture capture models these long wires as unidirectional which we expect to have a modestly negative impact on routability.

The architecture capture defines a set of custom SBs that model the statistics of how often each type of wire connects to each other type of wire. The actual 7-series devices all use a horizontal channel width of 124 wires and a vertical routing channel width of 190 wires. In contrast, our architecture capture can be targeted with different channel widths for experimentation; it will maintain the relative frequency and switch count of each wire type and the same 0.65 ratio of horizontal to vertical channel width. The CLB input pins can only be reached by length-1 and length-2 wires (both cardinal and diagonal); longer wires cannot connect to function block inputs. Our architecture capture models this connectivity, along with the local feedback connections that allow BLE inputs to be directly driven by BLE outputs in the same cluster. For DSPs and BRAMs, the AMD/Xilinx 7-series architecture has special local routing structures to improve connectivity

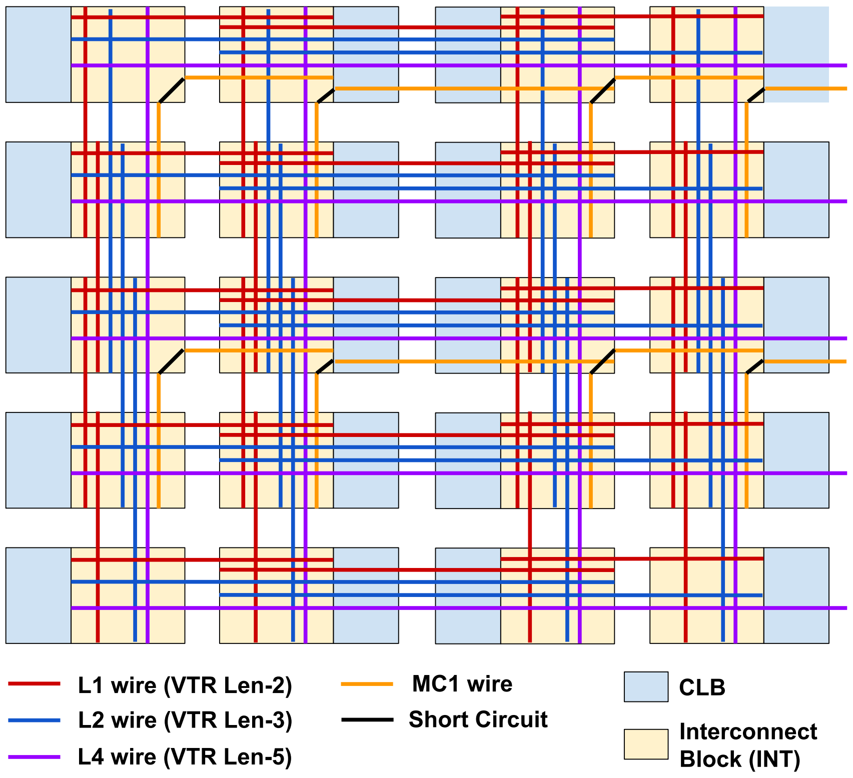


Fig. 8. Routing architecture of the AMD/Xilinx 7-series architecture.

between these blocks and general routing. In our capture, these structures are approximated using a high F_c values and full crossbars to work around routability issues to/from these blocks.

6.2.4 Timing Models. We use the Project X-Ray database [88] to obtain the timing information for the CLBs (including delays for LUTs, FFs, adder chains, local multiplexing) and different wire types. Similarly to the Stratix 10 architecture capture, we sum the delay of each routing wire type with that of its driving multiplexer and annotate the combined delay on the routing switch that drives each type of wire. The timing models of the included hard blocks (DSPs, BRAMs) are currently approximations based on the VTR comprehensive architecture. While the overall timing model is simpler than the Vivado timing model, we find that the CPDs it produce are broadly in line with those of Vivado.

6.2.5 Validation against Vivado Results. Table 7 shows the implementation results of six of the largest VTR benchmarks targeting our 7-series architecture capture. All results are at the actual 7-series channel width (W_v = 190 and W_h = 124) except the minimum routable vertical channel width and the designs are run on a 3.6 GHz Intel Xeon W-2123 server with 64 GB of RAM. The values in brackets in Table 7 are the VTR results divided by the corresponding value for AMD Vivado 2023.2 targeting the fastest speed grade of Kintex-7. Both VTR and Vivado use one CPU core, target an aggressive (unachievable) clock frequency to maximize design speed, and use their default optimization settings except VTR has the `--flat_routing` option described in Section 8.7 enabled. The results show that FF and RAM utilization is similar to Vivado, but that the number of LUT primitives is higher on four designs, and lower on two. Parmys mapped more multiplications

Table 7. Implementation Results of Six of the Largest VTR Benchmarks on Our 7-Series Architecture Capture

Benchmark	Resource Utilization								CPD (ns)		Runtime (s)		Min. W_v
	LUT Primitives		FFs		DSP Slices		18 Kb BRAMs						
bgm	26,073	(2.08×)	5,105	(0.88×)	22	(1.00×)	0	(—)	20.65	(1.32×)	1,086	(1.66×)	122
LU8PEEng	19,670	(1.18×)	6,224	(1.13×)	16	(1.00×)	45	(1.15×)	87.17	(1.12×)	1,224	(0.47×)	140
LU32PEEng	69,904	(1.28×)	19,668	(1.25×)	64	(1.00×)	153	(1.04×)	86.76	(1.08×)	5,718	(2.26×)	174
stereovision0	6,779	(2.28×)	11,497	(0.98×)	0	(—)	0	(—)	3.99	(0.88×)	5,25	(1.95×)	82
stereovision1	6,363	(0.39×)	10,315	(0.89×)	152	(—)	0	(—)	4.54	(0.70×)	461	(0.34×)	66
stereovision2	8,458	(0.84×)	15,659	(1.19×)	468	(1.90×)	0	(—)	14.58	(1.00×)	1,348	(3.16×)	80
Geomean	1.15×		1.04×		1.17×		1.09×		1.00×		1.25×		104

The ratios between brackets compare to Vivado results targeting a 7-series device. The ratio is omitted (—) when a resource is not used by Vivado and the geometric average is calculated excluding these data points. VTR 9 runtime is the summation of synthesis, packing, placement, and routing times. Vivado runtime is the summation of synthesis, physical optimization, placement, and routing times.

to DSP blocks than Vivado for the two designs (`stereovision1` and `stereovision2`) with lower LUT usage, and we attribute Vivado's lower LUT usage on the other four designs to the fact that its synthesis algorithms have been highly tuned for the 7-series. While the timing model is approximate in our architecture capture, the CPD for all circuits are within 32% of the Vivado value and the average CPD matches that of Vivado, indicating broadly reasonable delays. The runtime of VTR targeting this architecture averages 1.25× that of Vivado. This higher runtime is mainly due to the VTR packer, which performs general legality and routability checks for all clusters and is the most time-consuming stage of the VTR flow for the 7-series architecture (averaging 75% of the total). It is likely packing runtime could be reduced in the future by tuning the legality checks for Xilinx-style architectures. Finally, all circuits were successfully placed and routed with a minimum channel width less than that of the commercial 7-series architecture. On average, the benchmarks needed only 55% of the 7-series physical channel width to successfully route.

7 Logic Synthesis Enhancements

As shown in Figure 2, the logic synthesis stage of the VTR CAD flow generates a technology-mapped netlist from the **Register-Transfer Level (RTL)** description of a given circuit. This stage comprises several steps: RTL elaboration of Verilog or SystemVerilog to a netlist of operations, partial mapping of some operations to hardened FPGA structures (e.g., multipliers, adders, memories), general optimization of the remaining logic equations, and technology mapping to the types of resource available in the specified FPGA architecture. The technology-mapped netlist can then be passed to the physical implementation flow (packing, placement, routing) to determine the exact configuration of the FPGA resources to implement the given design.

VTR 9 supports multiple logic synthesis flows as shown in Figure 2. Odin-II provides intelligent partial mapping capable of performing hard logic inference and hard/soft tradeoff decisions (e.g., for multipliers) as to whether to map the logic to embedded hard blocks or the programmable soft logic [89]. However, the lack of support for many commonly used HDL features limits the set of designs that Odin-II can process without tedious and time-consuming recoding of the design to use only the supported language subset. The second logic synthesis option is the Titan flow [78] which uses the Intel/Altera Quartus synthesis engine, logic optimizer, and technology mapper. Then, it uses the `VQM2BLIF` tool to convert the output netlist to a format compatible with the rest of the VTR flow (BLIF). While this flow has large language coverage and better optimization quality, it limits the architecture exploration capabilities of VTR as it can only target Intel/Altera commercial FPGA architectures that use the same primitives (e.g., logic cells, multipliers, basic memories).

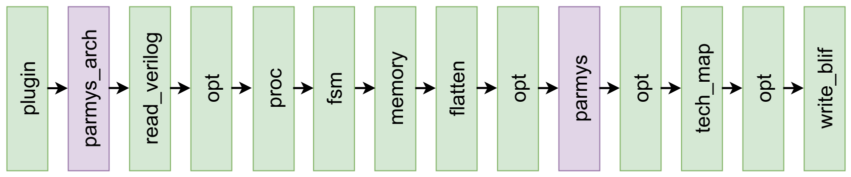


Fig. 9. Passes utilized in the Yosys flow (green: Yosys standard passes, purple: Parmys passes).

The third logic synthesis option is based on Yosys [45], an open source RTL synthesis framework with good language coverage and various passes that can be combined in different ways to customize the synthesis process. While some prior work has combined Yosys for logic synthesis with VTR for physical implementation for specific architectures (e.g., the Xilinx Artix-7 series in [58]), this is the first release to fully integrate (and regression test) Yosys within the VTR flow. VTR 9 augments Yosys with the new Parmys plug-in, which enables flexible mapping of design logic to the hard blocks described in the VTR architecture file, in keeping with VTR's goal of architecture flexibility. The Yosys + Parmys synthesis flow is the default in VTR 9.

Analogous to software compilers, Yosys consists of a frontend, a set of passes, and a backend. The frontend reads a specific input format (e.g., Verilog, SystemVerilog) and parses it into an internal RTL-level netlist called the **Register Transfer Level Intermediate Language (RTLIL)**. The RTLIL representation consists of RTL cells and gate cells. The RTL cells represent coarse-grained logic with multi-bit vectors for I/Os (e.g., multi-bit adders), while the gate cells represent fine-grained logic with single-bit I/Os (e.g., logic gates). Any algorithm implemented as a Yosys pass can then manipulate the circuit RTLIL representation. Yosys has numerous pre-defined passes, some of which call the ABC [90] tool to perform specific logic optimizations. When optimization is complete, the backend outputs the circuit in the desired format such as Verilog, BLIF, or the Electronic Design Interchange Format.

To create a synthesis engine that mixes the optimization flexibility and large language coverage of Yosys and the architecture awareness and intelligent partial mapping of Odin-II, VTR 9 adds two new plug-ins to Yosys. Figure 9 highlights our new passes added to Yosys in purple and the standard Yosys passes used in VTR in green. The `parmys_arch` pass is called early in the flow; it parses the VTR architecture file and extracts parameters relevant for logic synthesis such as BRAM sizes/capabilities, carry chain structures, LUT size, and other hard blocks that are available. This information is used to create appropriate Yosys module types and parameters so that Yosys can map to the specified architecture (e.g., targeting the proper LUT size) without any additional information provided by the user. The second new pass is `parmys`, which extracts and refactors the partial mapping stages from Odin-II in order to better leverage the various hardened elements on the target FPGA architecture. It determines the portions of the design to map to hard elements (e.g., adders, multipliers, memories, DSP blocks) and replaces them with black box modules to preclude further (de-)optimization by later Yosys passes. Parmys chooses which elements to map to hard blocks based on estimates of the relative efficiency in the hard block vs. being left in soft logic as well as balance estimates of whether the supply of some type of hard element in the target device is becoming limiting. The rest of the design is mapped to soft logic by Yosys. Finally, the VTR flow further optimizes the blif file output by Yosys by calling ABC's general logic optimization (`opt`) and technology mapping (`tech_map`) commands.

8 Physical Implementation Enhancements

VTR 9 introduces a number of enhancements to its placement and routing engines that add support for new architecture modeling features, improve QoR, and reduce runtime. Firstly, it implements an

Algorithm 1: The VTR 9 Initial Placement Algorithm

```
Input: $B$: list of clustered blocks, $G$: grid across $[x, y, layer]$ of possible locations in device
Output: Placement location in $G$ for each block $b_i$ in $B$
for each block $b_i$ in $B$ do
    $s_{b_i}$ = score($b_i$)    /* Higher scores for blocks with fewer location options */
    $H.push(b_i, s_{b_i})$    /* Insert $b_i$ in a max heap, $H$, sorted on $s_{b_i}$ */
end for
while $!H.empty()$ do
    $b_i = H.pop()$    /* Retrieve highest score block and remove it from $H$ */
    $(x_c, y_c, layer_c)$ = centroid location of the already placed drivers and sinks of $b_i$
    if $G[x_c, y_c, layer_c]$ is occupied or not compatible with $b_i$ then
        $(x_c, y_c, layer_c)$ = nearest empty and compatible location
    end if
    Place block $b_i$ at G$[x_c, y_c, layer_c]$
    $H.update\_scores(neighbours(b_i))$    /* Increase scores of blocks connected to $b_i$ */
end while
```

improved placement engine with a smarter initial placement, several placement perturbation moves that target the optimization of WL and/or timing, and an RL agent which dynamically selects the most suitable placement move to apply at different stages of the simulated anneal. The placement engine can also now accept user-specified floorplanning rules that constrain the placement of some design primitives to a certain region of the device. Secondly, the placement cost function is enhanced to consider the co-optimization of NoC quality metrics (e.g., latency and aggregate bandwidth) and conventional circuit metrics (e.g., CPD and WL) for designs implemented on FPGA architectures with hard packet-switched NoCs. Thirdly, both the placement and routing engines are extended to support 3D-stacked architectures. Finally, the VTR router can now perform inter- and intra-cluster routing in one unified step and route multiple nets (or net portions) in parallel. This section provides a summary of these new features and their effects on QoR and runtime.

8.1 Centroid Initial Placement

Placement is a critical step in the FPGA design flow that determines an optimized physical location within the device grid for each netlist block. Moreover, it is one of the most time-consuming steps in the FPGA CAD flow [78] and directly affects the final QoR metrics (e.g., WL and CPD). Similarly to earlier versions, VTR 9 uses a SA-based placement engine that starts with an initial solution and iteratively proposes perturbations (i.e., moves) to optimize it. It evaluates the change in the placement cost after each move to determine its acceptance. Initially, it operates at *high temperatures* allowing most of the moves to be accepted to be able to escape local minima. Then, as optimization progresses and temperature drops, the algorithm becomes more selective, reducing the probability of accepting moves that increase the placement cost. The quality of the final placement solution and the time to reach it are sensitive to the quality of the initial placement. Earlier versions of VTR created an initial placement by choosing a random legal physical location for each logical block in the packed netlist. This approach is fast to compute but has low quality as logical blocks connected by a net can initially be placed far from each other. Hence, the SA algorithm had to start with a high temperature and perform many perturbations to reach a high-quality placement.

In VTR 9, we implement a new initial placement algorithm (shown in Algorithm 1) to produce higher-quality solutions by attempting to place each block in proximity to the other blocks it is connected to, while maintaining a small runtime. The algorithm defines a score for each block that

Table 8. Centroid Initial Placement QoR and Runtime Relative to That of Random-Legal-Location Initial Placement (as Employed by VTR 8), Averaged across the Titan23 Benchmark Suite Using Five Random Seeds

Initial Place WL	Initial Place CPD	Final WL	Final CPD	Place Time
0.79	0.89	0.99	1.00	0.76

represents its priority to be placed. This score depends on multiple factors such as its floorplanning constraints (if any), whether it is a part of a carry-chain, and whether its connected blocks are already placed or not. Blocks are placed in descending order of their scores. For each block, we model its connections to other blocks as springs and following Hooke's law [91], we try to place it at the point of force equilibrium (i.e., the *centroid* location of its placed drivers and sinks). If this location is illegal (e.g., of an incompatible block type) or it is already occupied by another block, we search its surrounding area to find the nearest empty and compatible location. This new centroid initial placement can be computed quickly; it averages only 1.04 seconds of CPU time across the Titan23 benchmarks, which is negligible compared to the total placement time (>2,500 seconds). By placing connected blocks closer to each other, it produces an initial placement with 21% less wire and 11% less delay than a random starting point.

As SA has a better starting point, less hill climbing and fewer perturbations are required and we can start with a lower temperature; a lower temperature is also essential to prevent SA from destroying the initial placement. We calculate the standard deviation of cost over a small number of moves (all of which we reject to avoid degrading the placement); we empirically found a good initial temperature is 1/64 of this standard deviation [36]. This dynamic approach of computing the initial temperature is superior to using a fixed starting temperature as it automatically adapts to new cost functions, designs, and architectures. As a result of using a higher-quality initial placement solution, the SA initial temperature in VTR 9 is over 1,000× lower than that of VTR 8. On average across the Titan23 benchmarks compiled for the VTR Stratix-IV-like architecture using five different random seeds, this reduces the overall placement time by 24% while preserving or slightly improving QoR as shown in Table 8. VTR 9 also includes new command line options that allow an external tool to provide an initial placement that can be further optimized by the VTR annealer, enabling studies of hybrid placement algorithms such as the combination of analytic placement and annealing in [92].

8.2 Directed Placement Perturbations

As explained in the previous subsection, VTR uses a SA-based placement engine. Earlier versions of VTR explored the solution space only by randomly perturbing the locations of randomly selected blocks within a gradually shrinking region (i.e., *range limit*). To more efficiently explore the solution space, we propose a new set of move types (i.e., *directed moves*), each of which aims to optimize WL, CPD, or both metrics by intelligently choosing an alternative location of a given block based on anneal stage, current block location, locations of connected blocks, and estimated connection delays. Since these moves are performed thousands of times at each placement temperature, they should not only propose useful perturbations, but also be fast to compute. VTR 9 employs multiple move types which are summarized below and were developed and fully detailed in [93]:

(1) *Random*: Moves a block to a random location within a region around its current location.
(2) *Median*: Optimizes WL by moving a block to the closest legal location of a region defined by the median of the bounding boxes of the nets connected to it.

(3) *Centroid*: Optimizes WL by moving a block to the nearest legal location of the zero-force point when modeling the connections of the moving block as springs. This move is similar to the algorithm used in the centroid initial placement described in Section 8.1.
(4) *Weighted Centroid*: Similar to the centroid move but weights the force of each spring (i.e., connection) in the centroid calculation based on its timing criticality. This move optimizes both CPD and WL simultaneously.
(5) *Weighted Median*: Similar to the median move but weights each edge of the net bounding boxes in the median region calculation based on the timing criticality of the net terminal on that bounding box edge. This move optimizes both CPD and WL.
(6) *Critical Random*: Similar to random move but only tries to move blocks attached to timing-critical connections to improve CPD.
(7) *Feasible Region*: Moves a highly critical block to a random location inside the region which geometrically shortens its timing path [94] to improve CPD.

Most of these moves are combined with a bounding range limit that controls how large/disruptive the proposed perturbations are and gradually shrinks during the anneal. Early in the anneal this range limit is applied around the target location computed by each move type as described above. However, late in the anneal, the target location is used to determine a general direction towards which the block is moved within the bounding range limit around its current location. This strategy helps exploit the directedness of the moves without introducing quality oscillations.

8.3 Adaptive Placement Move Selection Using RL

The VTR flow needs to automatically adapt to different architectures and optimization goals, making it difficult to determine a fixed probability distribution for selecting from the various placement perturbations that will work well in all cases. Instead, the VTR 9 new placement engine, RLPlace, incorporates an RL agent that decides which directed move to use (out of the seven introduced in Section 8.2) and which block type (e.g., LB, DSP, BRAM) should be moved. The agent starts each placement with no prior knowledge and considers all available actions (i.e., moves to select) with equal probability. As optimization progresses, it learns that moving certain block types using certain directed moves is more beneficial (has a greater cost reduction per CPU unit time), and thus proposes the use of this pair (i.e., move and block type) more often. However, the agent still continues to propose less productive actions with a reduced frequency so that it keeps exploring the action space and can detect changes in productive move types during different stages of the anneal. The RL agent design we use was developed and described in more detail in [36] and further background on the application of RL to annealing-based placement can be found in [93, 95].

Combining this RL-based placement engine with the directed moves proposed in Section 8.2 improves both the placement QoR and runtime. Table 9 shows the geometric average WL, CPD, and placement CPU time for three different benchmark suites, with the relative values compared to VTR 8 in brackets. RLPlace consistently outperforms VTR 8 on all three benchmark suites, reducing placement CPU time by almost 2× while improving WL by 2–3% and CPD by 1–3%. Furthermore, RLPlace enhances the adaptability of the CAD flow so that it can efficiently target highly heterogeneous FPGAs with new features; for example, FPGAs with embedded NoC routers as studied in [36].

8.4 NoC Optimizations

As discussed in Section 4.1, VTR 9 supports architectural modeling of hardened NoCs in FPGA fabrics. The SA-based placement engine maps logical (netlist) routers to physical (in the device grid) routers at the same time that the other types of blocks are placed. The RL agent described in

Table 9. Geometric Average of RLPlace Post-Placement QoR and Placement Runtime for the Large (>10k Primitives) VTR, Titan23, and Koios Benchmark Suites, with Centroid Initial Placement

Benchmark Suite	**WL**		**CPD (ns)**		**CPU Time (s)**	
Large VTR	184,178	(0.98)	14.71	(0.98)	42	(0.51)
Titan23	2,572,680	(0.97)	16.20	(0.99)	1,433	(0.58)
Koios	1,948,209	(0.98)	8.09	(0.97)	375	(0.46)

The results are normalized to VTR 8 between brackets.

Section 8.3 automatically adapts and learns how frequently it should move logical NoC routers during the anneal. However, the placement cost function, packing attraction function, and the placement move generator are all enhanced [30, 83] as described in this subsection.

8.4.1 NoC-Related Cost Terms. We augment the placement cost function to co-optimize NoC-related metrics alongside the traditional ($C_{netlist}$) metrics of estimated programmable routing WL and CPD, as shown in Equation (1). The three new terms added to the cost function (C'_{bw}, C'_{lat}, and C'_{cong}) capture the aggregate bandwidth used over all NoC links, NoC latency, and NoC link congestion metrics. These metrics are normalized to equalize their magnitudes since bandwidth and congestion are on the order of 10^9 bits per second and latency is on the order of 10^{-9} seconds. The ω hyperparameter controls the weight of the NoC-related cost components; its default value is 0.6 which means NoC optimization is 60% as important as the total (timing plus wiring) netlist optimization components:

$$C_{Total} = C_{netlist} + \omega \times (C'_{bw} + C'_{lat} + C'_{cong}). \tag{1}$$

The NoC aggregate bandwidth cost term is the priority-weighted sum of the aggregate bandwidths of the set of all traffic flows T in the user design (see Section 4.1.2) as formulated in Equation (2). The aggregate bandwidth of a traffic flow T can be calculated as shown in Equation (3), where $BW(T)$ is the connection bandwidth of traffic flow T and N_{links} is the number of NoC links in the routed path of the traffic flow. Minimizing C_{bw} reduces the overall NoC bandwidth utilization by avoiding long travel paths for traffic flows (i.e., fewer physical NoC routers and links are traversed by a traffic flow):

$$C_{bw} = \sum_{T_i \in T} P(T_i) \times BW_{agg}(T_i). \tag{2}$$

$$BW_{agg}(T) = N_{links} \times BW(T). \tag{3}$$

The NoC latency cost term is the priority-weighted sum of two components over all traffic flows as formulated in Equation (4). The first component is the difference between the unloaded latency $Lat(T_i)$ and the user-specified latency constraint $LatConst(T_i)$, which directs the optimization towards satisfying latency constraints (i.e., ideally making $Lat(T_i) \leq LatConst(T_i)$). The second component is the absolute unloaded latency of the traffic flow, which aims to minimize the latency of the traffic flow in general. The two components are weighted using two empirically chosen hyperparameters, α and β. The unloaded latency of a traffic flow T is the latency of the traffic flow assuming no other traffic over the NoC. It can be calculated as shown in Equation (5), where N_{links} and $N_{routers}$ are the number of traversed links and routers on the routed path, and L_{link} and L_{router} are the link and unloaded router latencies, respectively.

$$C_{lat} = \sum_{T_i \in T} P(T_i) \times \Big(\alpha \times \mathtt{max}\big(0, Lat(T_i) - LatConst(T_i)\big) + \beta \times Lat(T_i)\Big) \tag{4}$$

$$Lat(T) = N_{links} \times L_{link} + N_{routers} \times L_{routers} \tag{5}$$

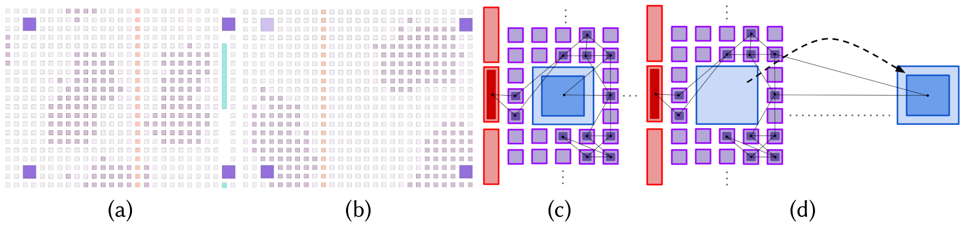


Fig. 10. The impact of NoC-aware packing and placement optimizations: (a) a design with three NoC components placed in compromise locations after conventional clustering, (b) the three NoC components kept separate during clustering and placed closer to their routers, (c) a router and the blocks connected to it, and (d) the same router moved to a new location, stretching its connections to other blocks.

The NoC congestion term is the sum of link bandwidth overutilizations across the set of all NoC links L as formulated in Equation (6). $LinkU(L_i)$ is the bandwidth utilization of link L_i. It can be computed as shown in Equation (7), where $1(T_i, L)$ is an indicator function specifying whether T_i is routed through L. The sum of all link bandwidth overutilizations is multiplied by a weighting factor, γ, that controls the importance of this congestion term relative to other NoC-related cost terms. The default values of α, β, and γ are 0.6, 0.02, and 0.25, respectively, giving the highest priority to meeting latency constraints and second priority to resolving congestion.

$$C_{cong} = \gamma \times \sum_{L_i \in L} \max\big(0, LinkU(L_i) - BW(L_i)\big) \tag{6}$$

$$LinkU(L) = \sum_{T_i \in T} 1(T_i, L) \times \; BW(T_i) \tag{7}$$

The placement engine updates these new NoC-related cost components only when the netlist primitives moved are logical routers. To update these components, all the traffic flows originating or terminating at the moved logical router(s) need to be re-routed to calculate their new aggregate bandwidth, latency, and congestion. VTR supports a variety of packet routing algorithms for mesh NoCs, including direction-ordered, XY, north-last, west-first, negative-first [96], and odd-even [97]. It also supports routing for non-mesh topologies with a minimal breadth-first search routing algorithm. The NoC placement optimization code is modular so that any new deterministic NoC routing algorithm can be easily added and used in the placement cost function calculation.

8.4.2 NoC-Aware Packing. VTR's packing algorithm tries to minimize the demand for routing wires by capturing low fanout connections inside clustered blocks. The packing algorithm first picks a primitive as the seed of a cluster and then adds other primitives that are connected to it through low fanout nets to the same cluster. However, to avoid cluster resource underutilization, the packing algorithm may use high fanout nets to find new candidate primitives to add to the cluster. This reliance on high fanout nets to infer logical connectivity can lead to sub-optimality in designs where modules communicate primarily through the NoC. For example, in Figure 3, modules M2 and M3 do not share any netlist connections except for high fanout nets such as clock, reset, and clock enable. If the packing algorithm uses high fanout connectivity to infer logical relevance between primitives in M2 and M3 and groups them into the same cluster, the placement engine needs to place clusters containing primitives that are connected to multiple NoC routers. As shown in Figure 10(a), since physical NoC routers are relatively distant from each other, clusters connected to multiple routers result in *compromise* placements where some logic is placed between two routers with relatively long connections to both.

To address this issue, VTR performs a breadth-first search on the primitive netlist with high fanout nets removed to group netlist primitives into disjoint *NoC components*. The primitives in a NoC component transitively connect to each other through the programmable routing fabric, while those in different NoC components are connected only via the NoC or high-fanout connections; designs that use the NoC to logically decouple modules will therefore contain multiple NoC components. We have modified the VTR packer so that by default it will not cluster primitives from different NoC components together. With this optimization, the placement engine can place NoC-attached clustered blocks close to their NoC routers, as shown in Figure 10(b), with a negligible increase in the number of clustered blocks.

8.4.3 NoC-Biased Centroid Move. Moving a router is more disruptive than moving most blocks, as they typically connect to many other fabric blocks and their legal locations are relatively distant from each other in the FPGA grid. Figure 10(c) shows a router that is connected to many fabric blocks. When the router is moved to a new location, as shown in Figure 10(d), the other blocks are still in their old locations, stretching the connections implemented using the programmable routing. To increase the probability of these blocks moving closer to the router's new location, we introduce a NoC-biased centroid move that adjusts the computed centroid location for a block toward the location of the router(s) associated with its NoC component. By gradually moving blocks of a NoC-attached module toward their corresponding routers, this move type facilitates optimization of entire NoC components.

8.4.4 Results. Table 10 summarizes the impact of VTR's NoC-aware packing and placement algorithms on the Hermes benchmark suite (see Section 5.3). For the most complex designs in this suite, namely the four large MLP benchmarks, the NoC optimizations in VTR 9 reduce the total used NoC bandwidth by 47% and the aggregate (summed over all traffic flows) latency by 37%, at no cost in placement time or programmable routing WL, but with a CPD increase of 5%. In these MLP benchmarks, the traffic flow bandwidths are low enough that there is no link congestion. For the synthetic benchmarks with known ideal NoC placements, NoC-aware algorithms show similarly large improvements in NoC aggregate bandwidth and latency. Unlike the MLP benchmarks, poor NoC placement or low-quality traffic flow routing can lead to congestion in these circuits. Enabling the congestion-aware NoC placement algorithm reduces congestion by 40% compared. NoC-aware optimization for these designs does not increase CPD, but increases the routed programmable WL by 8% and placement runtime by 17% compared to NoC-oblivious VTR. The peripheral interface benchmarks present complex traffic flow patterns that make congestion resolution challenging. Enabling VTR's NoC-optimization algorithms reduces congestion by 99%, with no CPD increase and a 9% programmable WL increase. Overall, the VTR 9 approach of co-optimizing the physical implementation of the logic and the NoC routers of a design works well; NoC quality metrics are significantly improved with modest impact on the design operating frequency and programmable routing usage.

8.5 Floorplanning Constraints

VTR 9 introduces flexible constraints that can be used to control the placement of some or all of a design, at various levels of detail. These constraints are useful for several purposes, including manually controlling the optimization of timing-critical logic, floorplanning a design into disjoint portions so that different members of a team can independently optimize a design and integrate it physically later, and re-using pre-placed IP modules for rapid compilation flows. As shown in Listing 6, primitives in the design are added by name to a *partition*; to make it easier to floorplan large parts of the design these names can include regular expressions (e.g., the * wildcard) that match entire hierarchies in the design. Each partition is assigned to a *region* on the device as

Table 10. QoR and Runtime Results of VTR 9 on the Hermes Benchmarks Suite, with Default NoC Optimization Settings and Odd–Even Packet Routing

Circuit	Place Time (s)		WL		CPD (ns)		Agg. BW (Gbps)		Agg. Latency (ns)		Congestion Ratio	
MLP1	4,462	(1.01)	12,251,216	(1.00)	9.91	(1.03)	12.33	(0.47)	12.34	(0.47)	0.0	(–)
MLP2	2,808	(0.99)	8,713,097	(1.00)	10.04	(1.09)	21.53	(0.51)	21.53	(0.51)	0.0	(–)
MLP3	2,814	(1.00)	8,547,047	(0.98)	9.60	(1.03)	20.45	(0.59)	20.45	(0.59)	0.0	(–)
MLP4	1,889	(1.01)	6,336,483	(0.97)	10.03	(1.04)	41.76	(0.56)	41.76	(0.56)	0.0	(–)
MLP Geomean	2,857	(1.00)	8,719,750	(0.99)	9.89	(1.05)	12.27	(0.53)	46.76	(0.63)	0.0	(–)
Synth. Geomean	405	(1.17)	563,083	(1.08)	7.19	(0.99)	15.2	(0.58)	411	(0.65)	0.78	(0.60)
Periph. Geomean	894	(1.11)	1,179,616	(1.09)	7.16	(0.99)	117	(0.30)	507	(0.44)	1.51	(0.01)

The numbers in brackets are the results normalized to VTR 9 with NoC optimizations disabled. The results for the MLP benchmarks are averaged over five different seeds while the synthetic and peripheral interface benchmark results are averaged over three seeds.

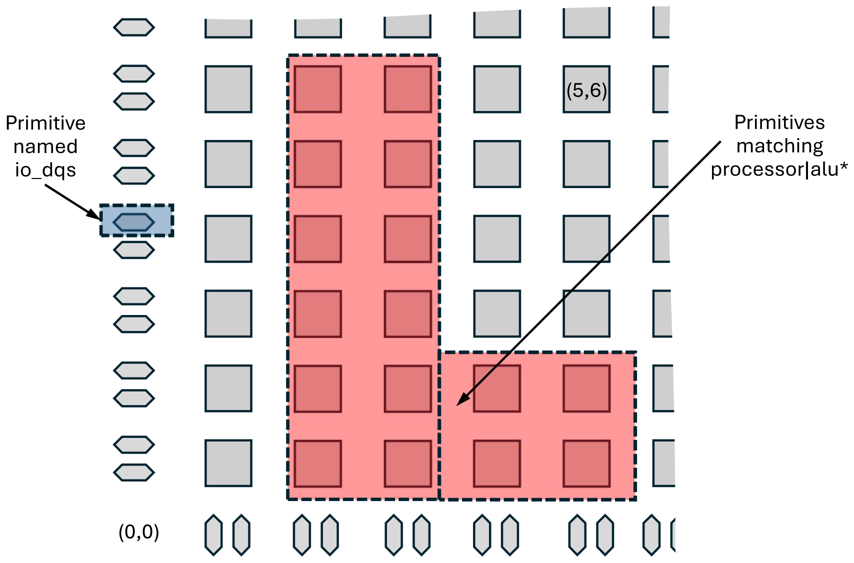


Fig. 11. Floorplan regions corresponding to Listing 6.

illustrated by Figure 11. A region is a union of rectangles (for a 2D device) or rectangular prisms (for a 3D device), which allows floorplanning to complex shapes like T-shaped or L-shaped areas or even disjoint areas. The regions can also be as specific as a single location on the chip, by specifying an area that covers only one tile on the chip, or even a single *subtile* in the case where there might be multiple blocks (e.g., IOs or PLLs) at a single (x, y, layer) location.

Floorplanning constraints are specified on design primitives but the VTR placement engine places clusters. This means that the packing engine needs to be aware of floorplan constraints as it creates clusters, or they will usually be impossible to place. Figure 12(a) shows an example floorplan with three regions (red, blue, and green); note that the red and blue regions partially overlap. As the packing algorithm groups primitives together into cluster-level blocks (LBs, BRAMs), it calculates a *feasible region* where the cluster could be placed by intersecting the floorplan regions of all the primitives within the cluster. If adding a primitive to the cluster would lead to an empty feasible region, the addition will be rejected as it would lead to an unplaceable cluster. Figure 12(b) depicts an example in which the cluster being created contains primitives from both the red and blue regions as well as unconstrained (white) primitives. Only primitives whose floorplan regions

```
<vpr_constraints>
   <partition_list>
      <partition name="red_region">
         <!-- Use wildcards to match everything in the processor's alu -->
         <add_atom name_pattern="processor|alu*">
         <!-- Constrain to an L-shaped region (union of two rectangles) -->
         <add_region x_low="2" y_low="1" x_high="3" y_high="6"/>
         <add_region x_low="4" y_low="1" x_high="5" y_high="2"/>
      </partition>
      <partition name="blue_region">
         <!-- Place the io_dqs primary input on a specific IO location -->
         <add_atom name_pattern="io_dqs">
         <add_region x_low="0" y_low="4" x_high="0" y_high="4" sub_tile = "1"/>
      </partition>
   </partition_list>
</vpr_constraints>
```

Listing 6. Example of VTR placement constraints file.

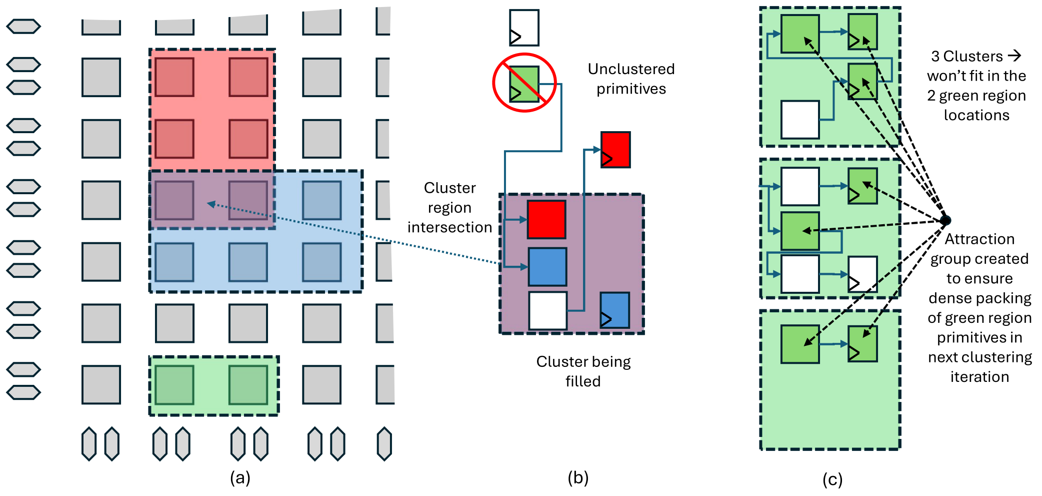


Fig. 12. Example of packing algorithm enhancements to respect floorplan constraints. (a) Three floorplan regions. (b) A cluster being created. Primitives are color-coded to match their floorplan region constraints; white primitives have no constraint. The intersection of the floorplan regions of its contained primitives is shown in purple and a primitive floorplaned to the green region will not be added to the cluster even if strongly connected to it. (c) The green floorplan region is overfilled, leading to the creation of an attraction group and a clustering retry.

overlap the resulting purple intersection of regions will be considered as candidates to be added to this cluster; the green register will not be considered despite being strongly connected to the cluster. If several clusters need to be placed within a floorplan region they could still exceed the number of placement locations within it, and therefore it can be beneficial to densely cluster primitives that have the same floorplanning region to maximize the chance they can be legally placed. On the other hand, forcing primitives with the same floorplan constraint to be packed as tightly as possible will tend to increase wire use by making less natural clusters and by leaving no room for other (unfloorplanned) primitives that may benefit from being in the same cluster due to their

Table 11. Geometric Average QoR on the 27 Synthetic Designs from the Hermes NoC Benchmarks, Normalized to the Unconstrained Placement Case

NoC Routers Locked	Floorplan-Guided Placement	WL	CPD	Place Time	NoC Agg. BW	NoC Cong.
✗	✗	1.00	1.00	1.00	1.00	1.00
✓	✗	1.03	1.00	0.86	1.00	1.00
✓	✓	0.91	1.00	0.79	1.00	1.00

Results averaged over three placement seeds.

connectivity to it. To balance these conflicting goals, the VTR packer takes an adaptive approach as illustrated in Figure 12(c). Initially it creates natural clusters where it will not add primitives that have little connectivity to the cluster, even if that leaves a cluster only partially filled. As detailed in [29] this reduces WL and CPD, but increases cluster count. Once a complete clustering is found, the packer checks for *overfilled regions* in which the clusters assigned to the region outnumber the tiles of the appropriate type within it. In Figure 12(c), for example, this natural clustering produces three clusters that contain primitives that must be placed in the green region in Figure 12(a), but the region contains only two tiles. For each overfilled region, an *attraction group* is created; it acts like a highly weighted virtual net connecting all the primitives assigned to the overfilled region so they now see a strong attraction to each other. Clustering is attempted again with these new attraction groups included in the clustering gain function in addition to the usual timing and WL terms, and all regions that were overfilled will now be densely clustered. Up to five attempts at clustering are allowed, but even difficult cases usually converge after two to three attempts.

After packing, clusters are placed so that they respect their floorplanning constraints at all times. As detailed in Section 8.1, blocks with fewer legal locations (due to tighter floorplan constraints) are placed first and must be placed within their floorplan region. Blocks for which there is no legal location given their constraints and previously placed blocks will be marked as initial placement failures. If there are any such failures, initial placement is retried with failed blocks placed first. Once initial placement is complete, the annealer optimizes the placement by iteratively moving blocks as described in Section 8.2. Moving blocks are checked against their floorplan regions and the move is aborted if they would be moved outside their regions. To reduce the number of aborted moves, which waste CPU time, most types of moves ensure legality by intersecting a block's target region with its floorplan region, ensuring that the proposed new location respects the floorplan.

To demonstrate the utility of these new floorplanning constraints, we use them to guide the placement of the 27 synthetic benchmarks from the Hermes suite of NoC-enabled benchmarks. In this subset of the Hermes suite, design modules communicate internally using programmable routing, but communicate with other large modules only over the NoC. We run VTR once with no constraints, allowing it to find a placement for both NoC routers and the design logic. We then run two constrained placement experiments. The first locks the routers down to the locations found in the first compile. The second not only locks the routers in place, but also floorplans each design module to a region that contains sufficient resources and the physical routers to which the module connects. Table 11 summarizes the results, geometrically averaged over all 27 designs and 3 placement seeds. Constraining the physical router locations saves 14% placement time as the placement engine can re-use its NoC placement results from the first compile, but does not improve result quality. Constraining the physical router locations and floorplanning the design logic achieves a larger 21% placement time reduction and reduces WL by 9% without degrading CPD, showing that well-chosen floorplan constraints can improve result quality.

8.6 3D Place and Route

The placement and routing engines have both been generalized to target any 3D-stacked device described in the architecture file. As discussed in Sections 8.2 and 8.3, the VTR SA-based placement engine uses various types of moves to alter the location of a block. We generalize each of these types of moves to operate in three dimensions; for example, the *centroid* move that previously proposed moving a block near the 2D-centroid (x_c, y_c) of its connections to other blocks [93] now attempts to move it near its 3D-centroid $(x_c, y_c, layer_c)$. The functions that estimate wiring and delay cost during placement are similarly updated to support 3D devices, and automatically adapt to the delay values and 3D interconnect style specified in the architecture file.

The routing engine is also updated in several ways. Firstly, the RR graph generator is generalized to add programmable routing switches and additional routing wires representing solder bumps or direct metal attachments between dice as specified by the architecture file. Secondly, before the net router begins, the router lookahead computation runs several all-destination Djikstra shortest path searches to profile the routing architecture and stores the results as a function of the routing node being examined and the 3D distance to the net sink. This enables fast 3D-architecture-aware lookups of the expected remaining wiring and timing costs as nodes are being examined by the subsequent routing path searches in the main routing algorithm. Thirdly, die crossing can be a choke point in 3D architectures where only a subset of the block output pins have programmable switches enabling them to cross dice, a style of architecture with attractive electrical properties [31]. Thus, we modified the router to sort the sinks of each net so that die-crossing connections are routed first for multi-fanout nets, ensuring the use of output pins with connectivity to the other die. The routing to other (same-layer) sinks of such a net then branches off the partial routing formed to reach the cross-layer sinks, which greatly improves router convergence.

To validate the QoR of VTR 9 when targeting 3D FPGAs, we evaluate a device with two identical FPGA fabrics stacked on top of each other in comparison to a 2D device with similar resource count and architecture (i.e., a single die that is twice as large) using the Koios benchmarks. We use the architecture from [21], which has Agilex-like DSP blocks and a Stratix-IV-like routing architecture. A key parameter in 3D-stacked architectures is the pitch between inter-die connections, as it determines the die-to-die interconnect density and has important manufacturing implications. We experiment with architectures under two different assumptions: a 5 μm pitch which represents state-of-the-art commercial 3D-stacking technology using direct metal attachment (i.e., hybrid bonding), and a more aggressive 1 μm pitch that is still in the research phase [98]. With a 5 μm pitch size, only 60% of the output pins of each LB can have programmable switches that enable connection to the other layer, while with a 1 μm pitch size all block output pins can connect to the other die. Inter-die connections have a delay of 75 ps [31], which is in the same range as same-die inter-block routing wires. For the scenario in which only 60% of the output pins can access the other die, we perform *light packing* where we constrain the VTR packer to use no more than 60% of the block output pins (using the command line option `--target_ext_pin_util`) to ensure there are always enough inter-die connections available during routing.

As shown in Table 12, when all block outputs have programmable switches to the other die, the CPD and WL are reduced by 3% and 2%, respectively compared to a 2D architecture with similar resources. Light packing increases the number of LBs needed by a design by 3% on average, but improves both 2D and 3D runtime and QoR by making the packing problem easier. Comparing the light packing results shows that a 3D architecture with only 60% of output pins connecting across layers also improves result quality vs. a 2D architecture, reducing CPD by 3% and WL by 4%. These early 3D results give us confidence that VTR 9 is optimizing for 3D devices well, but only scratch the surface of the architecture options that can be explored.

Table 12. VTR 9 Geometric Average Results over the Koios Benchmark Suite on 3D Architectures Normalized to a 2D Architecture with a Similar Number of Resources

	Default Packing		**Light Packing**	
	2D	**3D**	**2D**	**3D**
Inter-Die Connectivity	–	100% OPins	–	60% OPins
LB Count	1.00	1.00	1.03	1.03
Routed CPD	1.00	0.97	0.98	0.95
Routed WL	1.00	0.98	0.98	0.94
Runtime	1.00	1.02	0.73	0.8

8.7 Run-Flat: Unified Intra- and Inter-Cluster Routing

The VTR AIR is a PathFinder-based router with several enhancements to reduce CPU time using incremental routing techniques [65]. At its core, the PathFinder negotiation-based algorithm routes a design by iteratively ripping up and re-routing nets to resolve congestion [99]. VTR internally models the RRs within an FPGA as an RR graph in which nodes represent block pins and routing wires, while edges represent the programmable switches connecting them. Previous VTR versions perform routing in two stages: intra-cluster and inter-cluster routing. During the packing stage, intra-cluster routing is performed to route the nets between primitive pins and cluster pins using the local RRs in a cluster [28]. Then, after placement, the router performs inter-cluster routing only between clusters. This two-stage approach reduces the routing problem size (since the number of clusters is significantly less than the number of primitives) and as a result is simpler and faster; however, it has two major drawbacks.

First, the intra-cluster routing algorithm is not timing-driven, which may degrade the QoR. Second, having a 2-stage algorithm can lead to sub-optimal results on architectures where there is a switching network inside the cluster-level blocks that provides some flexibility, but not a complete crossbar between groups of cluster-level pins. In this case, the inter-cluster router is forced to use the cluster-level pins chosen by the packer (i.e., it cannot treat groups of pins as swappable), which means the flexibility of the intra-cluster routing network can be underutilized. Such architectures are becoming more popular as they reduce switch counts and wire loads compared to intra-cluster networks that use full crossbars between groups of pins [100] and they are employed in the most recent AMD and Intel FPGAs [87, 101].

To address these shortcomings, VTR 9 introduces the *run-flat* algorithm [32], which routes all connections between primitive pins in one step. While this improves QoR for FPGAs with sparse intra-cluster routing networks, it introduces several algorithmic and scalability challenges. Firstly, it expands the RR graph by adding intra-cluster pins and switches; the RR graph is the largest data structure in VTR so this impacts its memory footprint. Secondly, run-flat routes all connections between primitives, while the second phase of the two-stage router did not route the connections that are fully absorbed in a cluster. Thus, run-flat explores more nodes and edges as it routes nets to/from primitive pins within the cluster hierarchy rather than to/from pins on a cluster, which increases router runtime. Thirdly, we found that certain interconnect patterns within clusters create bottlenecks (which we denote as *choke points*) where traditional negotiated congestion approaches do not efficiently resolve congestion. Run-flat incorporates several new optimization techniques to address these challenges, as detailed below.

8.7.1 *Optimizing Memory Footprint.*

Previously, the RR graph generator considered only the target FPGA architecture; it created nodes for every routing wire and I/O pin in the device and edges between those that can be connected via programmable switches. To enable run-flat, the straightforward approach would be to add nodes and edges corresponding to all the elements

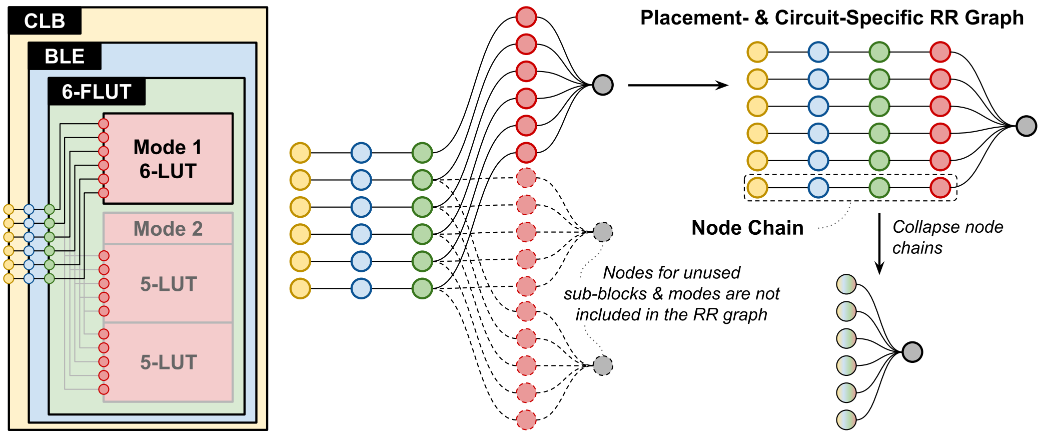

CLB
BLE
6-FLUT
Mode 1
6-LUT
Mode 2
5-LUT
5-LUT
Placement- & Circuit-Specific RR Graph
Node Chain
Collapse node chains
Nodes for unused sub-blocks & modes are not included in the RR graph

Fig. 13. An RR graph optimization example for unified inter- and intra-cluster routing. The intra-cluster part of the RR graph is generated after placement so that the usage of each physical cluster is known, and only the RR nodes and edges needed within each cluster are created. Fanout-free node chains are then collapsed into single nodes.

within each cluster for all its hierarchical levels, as described in the input architecture file. The RR graph would be further expanded to include all the operating *modes* of each level of hierarchy. For example, as illustrated on the left side of Figure 13, a fracturable 6-LUT has one mode with a single 6-LUT and another mode with two 5-LUTs sharing all inputs. The generated RR graph would include 16 nodes corresponding to the inputs of a 6-LUT and two 5-LUTs although only one of the two modes could be used for each fracturable LUT. Run-flat reduces the RR graph memory footprint in two ways. First, building of the intra-cluster portion of the RR graph is deferred until after placement. At this point, the portion of each cluster-level block that is used and its mode of operation (for each hierarchy level) is known and hence only the nodes and edges corresponding to these used modes are included in the RR graph, as shown at the top right of Figure 13. For the Titan23 benchmarks targeting the VTR Stratix-IV-like architecture, this reduces the number of nodes and edges in the RR graph by 8× and 5×, respectively. The second technique further simplifies the RR graph by collapsing chains of nodes (and the edges connecting them) that are unnecessary to evaluate whether a routing is legal or not. Directly building the intra-cluster RR graph from the architecture description results in an input pin and output pin per hierarchy level inside a cluster. However, often these pins have only one fanout, and hence a fanout-free-chain of such nodes can be replaced by a single node with delay equal to the total delay of the chain, as illustrated at the bottom right of Figure 13. Any routing which does not overuse the single remaining node is guaranteed to have a direct mapping back to the chain. Chain collapsing reduces the RR graph size by 15% for the largest seven VTR benchmarks on the VTR comprehensive architecture.

8.7.2 Optimizing Runtime. AIR uses a directed path search where the cost of a RR node is a function of its *known cost* from the source of the connection in addition to its *predicted (i.e., lookahead)* cost to the sink of the connection [65]. The router lookahead must automatically adapt to the target routing architecture to maintain VTR's flexibility, and to keep the path search efficient, it should accurately predict the future cost to reach the target sink from the current node. However, it must also be fast to compute and memory-efficient as it will be called every time a new node is explored by the router (i.e., pushed to the heap). The two-stage router in VTR uses a Dijkstra-flood algorithm to create a delay and congestion cost table storing the predicted cost to reach a given

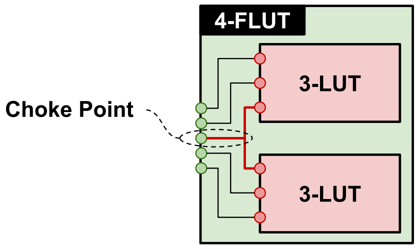


Fig. 14. A choke point example in a fracturable LUT.

sink based on the type of the current node and the its relative distance (Δx, Δy, $\Delta layer$) to the sink. In addition to the inter-cluster lookahead table, run-flat also constructs an additional intra-cluster lookahead table for each cluster type. This table stores the predicted delay and congestion cost to reach a target sink from a node within the same cluster and is only used when the path search enters the destination cluster. When the path search is still outside of the destination cluster, run-flat approximates the intra-cluster lookahead cost by using the pre-computed minimum predicted cost for the destination cluster type. Across different FPGA architectures, this enhanced hierarchical lookahead reduces CPU time for run-flat by 64–83% compared to using the original VTR lookahead.

8.7.3 Resolving Choke Point Congestion. The primitives and interconnect inside a cluster are often finely tuned by FPGA architects to efficiently implement some use cases. Some of these structures have only a few legal routing solutions, making legality more challenging to achieve using traditional congestion negotiation techniques. Figure 14 shows an example of a fracturable 4-LUT that can be split into two 3-LUTs with one shared input. Only one of the five inputs of the fracturable LUT can reach both 3-LUTs, so any successful routing must map the net that fans out to both 3-LUTs on this pin. Because negotiated-congestion-based algorithms route one connection at a time, they lack a net-level view of the problem, leading to poor convergence. In our experiments, when using run-flat with traditional congestion negotiation, over half of the designs targeting fracturable LUTs fail to find a legal routing at all. We solve this problem in a general way by finding *choke points* in the intra-cluster routing where fanout to multiple destinations is possible, and biasing the router to use these choke point nodes preferentially for nets with matching fanout needs. More specifically, at the start of routing, we identify potential net choke points where a net has more than one sink inside a cluster. Next, a reachability analysis in the RR graph determines the number of sinks that can be reached from each cluster input pin. Each RR graph node that can reach multiple sinks is considered a possible choke point cp, for which we store the number of reachable sinks $N_{sink}(cp)$. During routing of connections associated with net choke points, we divide the usual congestion cost of a node n by $2^{N_{sink}(n)}$ when n is in the set of node choke points. This leads to faster legality convergence as it provides the router with a whole-net view of each connection, biasing it to use pins that can reach multiple sinks for multi-fanout nets.

8.7.4 Results. Table 13 evaluates run-flat on three different architectures: (1) the Stratix-IV-like architecture capture which has a full crossbar in LBs (an overestimation of the intra-cluster connectivity provided by the commercial Stratix IV architecture) and a partial crossbar in BRAMs, (2) the VTR comprehensive architecture which has a 50% depopulated crossbar inside LBs and no crossbars in the hard blocks, and (3) our 7-series-like architecture capture (Section 6.2). The full crossbar in the LBs of the Stratix-IV-like architecture is more friendly for the two-stage router, as all the LB inputs can be modeled as equivalent (i.e., swappable). However, the BRAM input pins cannot be modeled as equivalent due to the partial crossbar, so the two-stage router cannot fully

Table 13. Run-Flat Results on Stratix-IV-like, VTR Comprehensive, and 7-Series-Like Architectures Relative to the Two-Stage Router

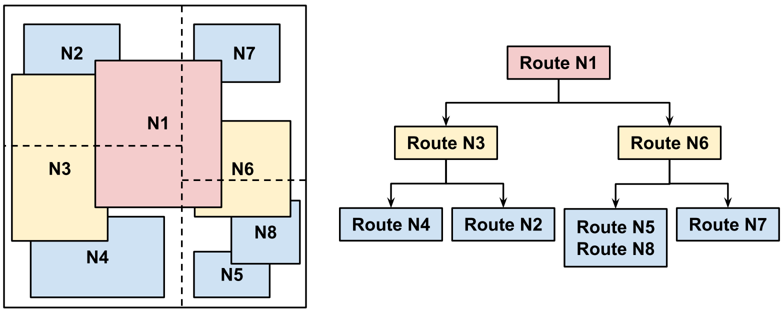
	Stratix-IV-Like	**VTR Comp.**	**7-Series-Like**
Benchmarks	Titan23	VTR	VTR
LB Local Routing	Full xbar	50% xbar	Irregular
Hard Block Local Routing	Partial xbar	None	None
Min. Channel Width	(Fixed $W = 300$)	0.86	(Fixed $W_v = 190$)
Routed WL	0.94	0.88	0.89
Routed CPD	0.99	0.98	0.92
Route Time	1.36	2.70	1.84
Total Time	1.16	1.29	1.16
Max. Memory	2.84	1.26	1.21

exploit it. On this architecture, run-flat reduces the average WL by 6% and CPD by 1% over the Titan23 benchmarks. The benefits are more pronounced with the VTR comprehensive architecture since the LBs also have depopulated crossbars. On this architecture, run-flat reduces WL by 12% and CPD by 2% on average across the VTR benchmarks. The 7-series has (non-crossbar) local routing in its LBs which the flat router can exploit much better than the 2-stage router, leading to an 11% reduction in WL and an 8% CPD reduction. Run-flat increases memory footprint to 1.21× - 2.84× that of the 2-stage router, with the smallest overhead for the 7-series-like and the largest for the Stratix-IV-like architecture. Run-flat increases route time to 1.36× - 2.7× that of the 2-stage router, but this translates to a more limited overall VTR flow runtime increase of 16–29%. These increases are due to the larger RR graph and larger number of connections to route within it, and hence run-flat presents a quality-runtime tradeoff. For architectures with partial local interconnect flexibility, the QoR gains generally justify the additional runtime.

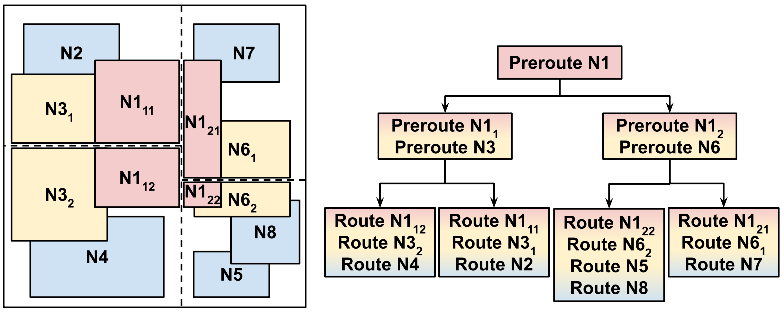
8.8 Parallel Router

In previous VTR versions, the routing engine (AIR) was single-threaded, routing one connection at a time. VTR 9 introduces two parallel routers (*baseline* and *net-decomposing*) that use *spatial bi-partitioning* to route different nets or different portions of the same net in parallel. Spatial partitioning for parallel routing has been previously used in several works [102–104]. A key differentiator of the VTR 9 parallel routers is that as they are built on top of AIR [65], and hence inherit its incremental techniques that greatly reduce the number of operations performed during routing. In addition, the net-decomposing router uses a new technique to sub-divide nets into components to extract more spatial parallelism [105].

8.8.1 Baseline Parallel Router. The first router is a baseline implementation of spatial bi-partitioning, where the device grid is recursively divided into two regions by choosing partition lines. After the nets crossing the partition line are routed, the sides of the partition line are now decoupled from each other in terms of cost updates, and nets on each side can be routed in parallel. An example of the parallelism exploited by the baseline router is shown in Figure 15(a), with the bounding box of each net illustrated on the left. The algorithm chooses cutlines to recursively divide the FPGA grid; this forms a partition tree (shown on the right side of Figure 15(a)) in which all nodes in the same level of the partition tree can be routed in parallel as they consist of nets which do not overlap and hence will not compete for the same RRs. The position of the cutline at each level is chosen to minimize expected execution time (i.e., the sum of routing time of the parent partition node and the slower of its two children), where the runtime of a partition node is estimated as proportional to



(a) Baseline



(b) Net-decomposing

Fig. 15. Spatial partitioning and the resulting partition trees for both parallel routers. All nodes (sets of nets) at the same level of a partition tree can be routed in parallel.

the number of connections (sinks) it must route. In Figure 15, net $N1$ would be routed first at the L1 partition level using one thread. The nodes in the next level of the partition tree (L2) can be routed in parallel, so nets $N3$ and $N6$ would be concurrently routed on two threads. Similarly, the L3 level of the partition tree contains four nodes and thus allows four threads to route nets in parallel.

8.8.2 Net-Decomposing Router. FPGA designs almost always contain some *high-fanout nets* that span a large area of the chip and contain many connections. Unfortunately, this means that they usually cross the first partition line and take significant time to route, and therefore they limit the parallel speedup attainable by assigning different partition tree nodes to different cores. To achieve higher speedups, VTR 9 includes a *net-decomposing* parallel router as shown in Figure 15(b). When a medium or high fanout net crosses a partition line, this router *decomposes* it into portions that will be routed at different levels of the partitioning tree. In the Figure 15(b) example, net $N1$ crosses the first cutline. Instead of routing all its connections in the root partition node on one processor, the net-decomposing router chooses a small subset of its sinks to pre-route at the first partition level. The remaining sinks are divided into two groups (those on the left and the right of the first cutline), and their routing is completed in parallel in two partition nodes in the L2 partitioning level. The left and right nodes in the second partitioning level are passed the portion of the pre-routing of net $N1$ that lies within the left and right spatial partitions, respectively. They can complete the routing of the remaining sinks of net $N1$ in parallel by each branching off only from the portion of the pre-routing they were passed, which is guaranteed to be spatially disjoint. If a net has enough sinks, this decomposition process continues further; additional sinks of net $N1$ are pre-routed on

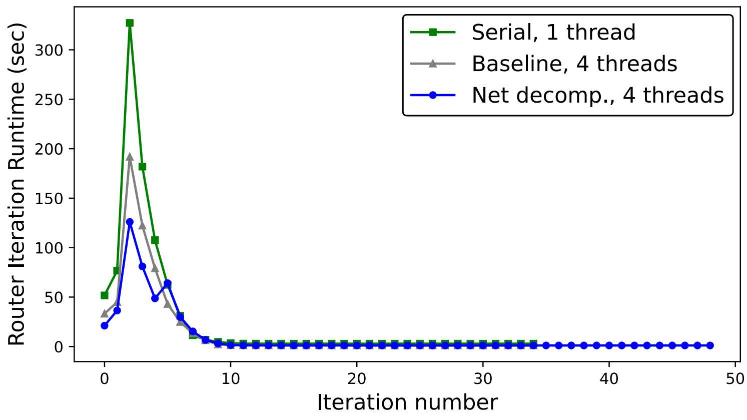


Fig. 16. Runtime per routing iteration for the LU230 circuit from the Titan benchmarks.

two cores in the second partition level (pre-route $N1_1$ and $N1_2$) and the remaining sinks are divided into four groups that are routed in parallel in the L3 partition level ($N1_{11}$, $N1_{12}$, $N1_{21}$, and $N1_{22}$).

8.8.3 Per-Iteration Tasks. After routing nets in each iteration, the VTR router has to perform additional tasks such as total WL computation, RR node historical congestion cost updates, and timing analysis. Since the AIR router keeps most of the routing results from previous iterations, there is very little time spent routing nets in the final iterations. This makes the runtime spent in per-iteration tasks significant. These operations are also now parallelized in VTR 9. Parallel timing analysis was previously implemented in [106] but was disabled due to non-deterministic results; we have made it deterministic and re-enabled it as part of the engineering work for the parallel router.

8.8.4 Results. The parallel router is evaluated on a machine running Ubuntu 22.04 on an AMD 7950X3D processor[1] and 96 GB of RAM. We used the Titan benchmarks with the default VTR settings in the `titan_quick_qor` task for evaluation. Figure 16 shows the runtime per iteration for the serial router, the baseline parallel router with four threads, and the net-decomposing router with four threads on a representative design (LU230) from the Titan benchmark suite. The AIR algorithm (on which all three routers are based) minimizes the number of connections that are re-routed each iteration; only congested or timing-compromised portions of routed nets are ripped up in each routing iteration. Hence, most of the work is performed in the earlier routing iterations, and in these iterations the net-decomposing router is approximately 3× faster than the serial router. Later iterations do much less work but still consume non-negligible time; the parallel routers are still slightly faster in this regime due to the per-iteration operations being parallel, but the speedup over the serial router is smaller.

Table 14 summarizes the overall runtime and quality of the different routing algorithms for both the two-stage and flat (see Section 8.7) cases. For two-stage routing, the baseline parallel router achieves a 2.1× speedup at eight threads with virtually no loss in QoR. The net-decomposing router has a speedup of 2.4×, albeit with 2.1% worse CPD and 0.7% higher WL due to the additional constraints imposed on the router by net decomposition. The results for flat routing are similar. The baseline parallel router with eight threads achieves a 2× speedup over the serial router with no effect on WL and a 1.5% increase in CPD, while the net-decomposing router achieves a 2.2× speedup but degrades CPD and WL by 2.3% and 0.9%, respectively.

[1]The processor has eight cache and eight frequency cores. VTR processes were pinned to the cache cores.

Table 14. Runtime and Quality Comparison of Serial and Parallel Routers with Eight Threads, Geometrically Averaged over the Titan Benchmarks and Normalized to the Serial Router with the Same Routing Method

Router	Per-Iteration Tasks	Net Routing	Runtime	CPD	WL
Two-Stage	Serial	Serial	1.000	1.000	1.000
Two-Stage	Parallel	Serial	0.757	1.000	1.000
Two-Stage	Parallel	Parallel (Baseline)	0.468	1.000	1.000
Two-Stage	Parallel	Parallel (Net-Decomposing)	0.421	1.021	1.007
Flat	Serial	Serial	1.000	1.000	1.000
Flat	Parallel	Serial	0.801	1.000	1.000
Flat	Parallel	Parallel (Baseline)	0.503	1.015	1.000
Flat	Parallel	Parallel (Net-Decomposing)	0.464	1.023	1.009

9 Software Engineering and Developer Utilities

VTR is a large open source project, and therefore requires considerable software engineering and documentation work to enable contributions and effective collaboration between various teams from both academia and industry. In the following sections, we list some of the features added in VTR 9 to improve its usability and software engineering practices.

9.1 VTR Utility and API Documentation

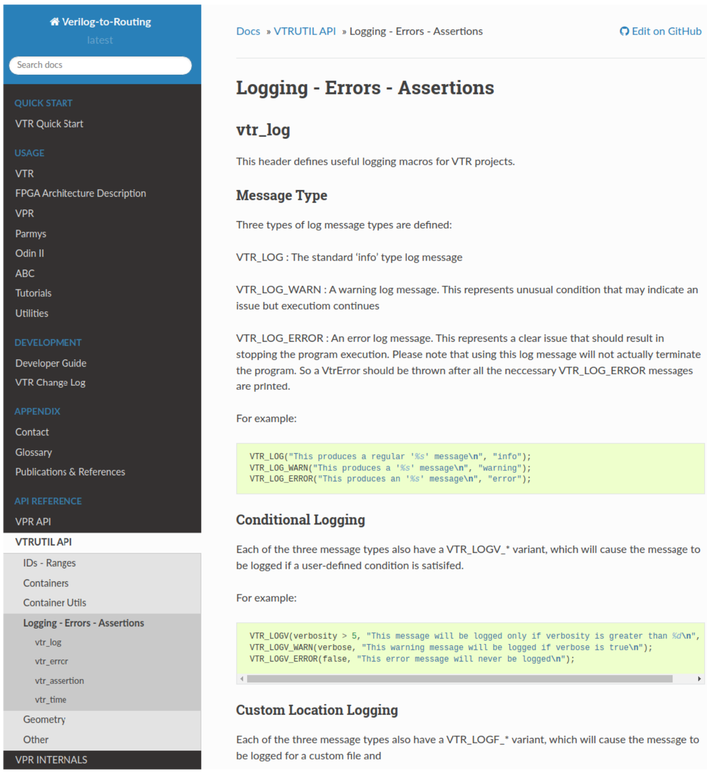
The VTR project contains several custom data types (e.g., N-dimensional matrices, bidirectional maps, and strongly typed identifiers) and utility functions (e.g., for logging, assertions, and runtime measurement). It also includes useful APIs for parsing, querying, and dumping key data structures such as RR graphs, circuit netlists, FPGA tile grids, and so on. For VTR developers, using these utilities and APIs can enhance their productivity and lead to a more maintainable code base with consistent program output formats. However, due to a lack of documentation, developers often did not know of the existence of these utilities or were unaware of how to best use them. VTR 9 adds extensive web-based documentation that is automatically built from Doxygen comments in the code base; Figure 17 shows an example. This documentation helps developers efficiently explore the available utilities and APIs, which is increasingly important as the number of contributors (currently over 150) to the VTR project continues to increase.

9.2 Batch VTR Runs Using SLURM

In many cases, evaluating new FPGA architecture features or CAD algorithms requires launching a large number of VTR runs using multiple seeds and different benchmark suites. These experiments consume significant CPU time and memory and are ideally dispatched on powerful compute clusters. For this reason, VTR 9 includes an interface to the open source SLURM job scheduler [107], which is widely used in cloud platforms such as Google Cloud and the Digital Research Alliance of Canada [108]. This interface allows batching many VTR runs using different benchmarks, FPGA architectures, and CAD parameters for transparent parallel dispatching on a cluster of servers.

9.3 Graphical Debugging of Optimization Algorithms

Debugging the behavior of a new CAD algorithm during its implementation and tuning can be a major challenge. These algorithms typically involve many operations/steps (e.g., block moves in SA-based placement), and could have not only bugs in their implementation but also flaws in the high-level algorithmic logic (e.g., optimization cost function). In many cases, these problems are only observable in certain benchmark designs or in later stages of an algorithm and therefore are difficult to investigate using conventional techniques such as code debuggers and logging. To



Logging - Errors - Assertions

vtr_log

This header defines useful logging macros for VTR projects.

Message Type

Three types of log message types are defined:

VTR_LOG : The standard 'info' type log message

VTR_LOG_WARN : A warning log message. This represents unusual condition that may indicate an issue but execution continues

VTR_LOG_ERROR : An error log message. This represents a clear issue that should result in stopping the program execution. Please note that using this log message will not actually terminate the program. So a VtrError should be thrown after all the neccessary VTR_LOG_ERROR messages are printed.

For example:

```
VTR_LOG("This produces a regular '%s' message\n", "info");
VTR_LOG_WARN("This produces a '%s' message\n", "warning");
VTR_LOG_ERROR("This produces an '%s' message\n", "error");
```

Conditional Logging

Each of the three message types also have a VTR_LOGV_* variant, which will cause the message to be logged if a user-defined condition is satisifed.

For example:

```
VTR_LOGV(verbosity > 5, "This message will be logged only if verbosity is greater than %d\n",
VTR_LOGV_WARN(verbose, "This warning message will be logged if verbose is true\n");
VTR_LOGV_ERROR(false, "This error message will never be logged\n");
```

Custom Location Logging

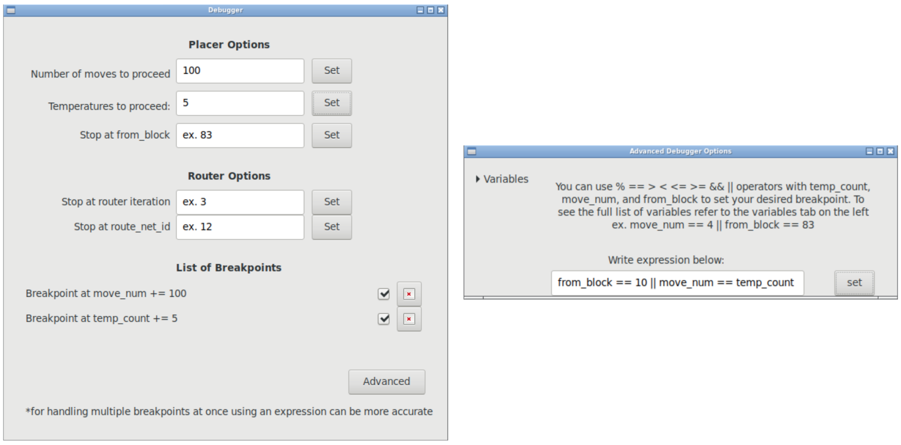
Each of the three message types also have a VTR_LOGF_* variant, which will cause the message to be logged for a custom file and

Fig. 17. The web-based documentation of VTR utilities and APIs.

facilitate algorithm debugging and tuning, we add a new feature to the VTR GUI that allows users to set *algorithmic breakpoints* in the placement and routing engines to pause execution when a specific condition is met and visualize the current optimization state. For example, the user can pause the placement optimization after a specific number of moves, at specific annealing temperatures, or when a specific block is moved, as shown in the left side of Figure 18. Similarly, the user can pause the router after a specific number of router iterations or when routing a specific net. The user can set more complex conditional breakpoints by combining various conditions with logical operations as shown in the right side of Figure 18.

9.4 User-Guided Placement Optimization

As discussed in Section 8.2, smart placement moves can greatly enhance the placement quality and reduce runtime. VTR 9 allows users to interactively propose placement perturbations in the GUI; this can be used in conjunction with the algorithm breakpoints discussed in Section 9.3 to test new placement move ideas. As shown in Figure 19, the graphical interface allows the user to propose relocating a certain block (specified by block ID or name) to a new grid location, evaluate the impact on the placement cost (estimated WL and CPD), and decide whether to accept or reject the move. This enables CAD developers to efficiently experiment with various move strategies, evaluate their effectiveness, and understand the reasons if they do not work as expected.



Debugger

Placer Options

Number of moves to proceed: 100 [Set]

Temperatures to proceed: 5 [Set]

Stop at from_block: ex. 83 [Set]

Router Options

Stop at router iteration: ex. 3 [Set]

Stop at route_net_id: ex. 12 [Set]

List of Breakpoints

Breakpoint at move_num += 100 ✓

Breakpoint at temp_count += 5 ✓

Advanced

*for handling multiple breakpoints at once using an expression can be more accurate

Advanced Debugger Options

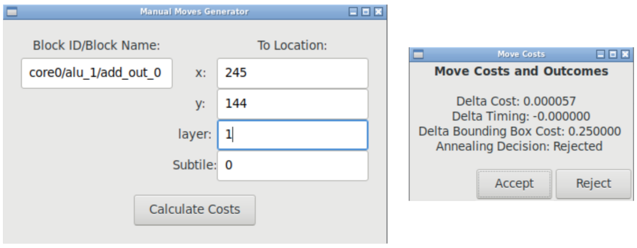
Variables

You can use % == > < <= >= && || operators with temp_count, move_num, and from_block to set your desired breakpoint. To see the full list of variables refer to the variables tab on the left
ex. move_num == 4 || from_block == 83

Write expression below:

from_block == 10 || move_num == temp_count [set]

Fig. 18. The VTR graphical debugging interface allowing users to specify simple breakpoints during certain steps of the placement and routing optimizations (left) or advanced breakpoints using variable names and logical expressions (right).



Manual Moves Generator

Block ID/Block Name: core0/alu_1/add_out_0

To Location:

x: 245

y: 144

layer: 1

Subtile: 0

Calculate Costs

Move Costs

Move Costs and Outcomes

Delta Cost: 0.000057
Delta Timing: -0.000000
Delta Bounding Box Cost: 0.250000
Annealing Decision: Rejected

Accept Reject

Fig. 19. Interactive user-guided placement optimization. The VTR GUI now allows the user to propose a certain move (left), evaluate its effect, and decide to accept or reject it (right).

9.5 Continuous Integration and Testing

The VTR project has become a key enabler for FPGA architecture and CAD academic research and the core of several commercial CAD systems [109–111]. Therefore, thoroughly testing its many flows, supported architectures, and optimization algorithms has become ever more important for maintainability and scalability. To this end, VTR 9 greatly expands test coverage and enhances test automation. It includes both unit tests that validate specific classes and sub-algorithms, and system-level tests that check the entire flow completes and achieves the expected QoR on various benchmark sets, architectures and use cases. Other tests also automatically check compatibility with a wide range of compilers and verify that no memory access errors/bugs are flagged by valgrind or code sanitization tools. All of these tests are launched automatically on each pull request to the main branch and must pass before a change can be merged. The tests use a combination of GitHub-hosted and Google-Cloud-hosted machines running in parallel so that over 500 CPU hours of tests are completed in 6 hours.

10 Overall Flow Result Quality

All results were collected on an unloaded system with two Intel Xeon gold 6146 CPUs (12 cores each with a base frequency of 3.9 GHz) and 768 GB of RAM. VTR was compiled with its default build settings: GCC 9.4 with full optimization (-O3) with inter-procedural optimization and profile-guided optimization using a profile generated from single runs of the `stereovision1` and `neuron` benchmarks. All experiments are run with three different placement seeds for each circuit and the results averaged to reduce CAD noise. In Sections 10.1 and 10.2, VTR is run on a single core (serial mode), while Section 10.3 explores the impact of using multiple CPU cores.

10.1 Logic Synthesis Evaluation with Koios Designs

Firstly, we evaluate the effect on QoR of using Parmys in comparison to Odin II for logic synthesis in VTR. In this experiment, the output of both logic synthesis flows is consumed by the same physical implementation flow with all the enhancements described in Section 8. We use 37 out of the 40 Koios benchmarks for this evaluation; the three excluded benchmarks are written in SystemVerilog, which is not supported by Odin-II. We target the `k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml` FPGA architecture [35] with a channel width of 300. In this architecture, the logic, RAM, and DSP blocks are Intel-Agilex-like but the routing architecture is similar to Stratix IV and delays come from COFFE [112] simulations of a 22 nm technology node.

Table 15 shows the key runtime and QoR metrics for Parmys, with the relative value of each metric compared to Odin-II in brackets. Odin-II failed to synthesize 14 of the 37 designs due to its limited Verilog language support.[2] On the other hand, Parmys can successfully process all the Koios designs. Parmys reduces synthesis time by 33% on average; while not shown in the table, it also reduces the memory utilization of synthesis by 12%. Parmys also improves quality metrics, primarily due to its better area optimization. The number of primitives required to implement a design drops by 11% on average, leading to 7% smaller auto-sized FPGAs (i.e., fewer device tiles). Routed WL and CPD are reduced by 5% and 4% on average, respectively.

10.2 Full Flow Evaluation

In this subsection, we evaluate the results of the overall VTR 9 flow in comparison to the previous VTR 8 release using the VTR and Titan23 benchmark suites.

10.2.1 VTR Benchmarks. Table 16 shows the results of the VTR 9 flow on the large (>10k primitive) VTR benchmarks targeting the VTR comprehensive architecture. The flow in this case searches for the minimum channel width for each design and then routes with a low stress channel width 30% above the minimum; all the quality and runtime statistics are from this low stress routing run. The overall flow time, resource usage, WL, and CPD are all very similar to those of VTR 8. The runtime of the placement and routing engines in VTR 9 is reduced by 24% and 21% vs. their VTR 8 equivalents, but on these smaller designs much of the runtime is spent in synthesis and in generating architecture data structures such as the RR graph, so there is only a 7% overall runtime improvement. Memory footprint has increased by 16% but since the VTR designs are of moderate size the absolute memory footprints are still very reasonable, with the largest design requiring 2.24 GB of memory.

10.2.2 Titan23 Benchmarks. The Titan benchmarks are larger and more complex than the VTR benchmarks, making them a better suite for evaluating the algorithmic enhancements in VTR 9. We target VTR's Stratix-IV-like architecture capture with a channel width of 300, which roughly matches the commercial device. Note that the Titan flow uses Quartus for logic synthesis, so only the physical implementation portion of the VTR 9 improvements affect these results. Table 17 shows that VTR 9 requires only 51% of the place time and 51% of the route time of VTR 8, leading

[2]The VTR 8 release had even more limited Verilog language coverage, so it cannot process most of the Koios designs and hence a quality comparison to VTR 8 cannot be made.

Table 15. Parmys Results on the Koios Benchmarks (Normalized to Odin-II between Brackets) Averaged over Three Different Placement Seeds

Circuit	Synth Time (s)	VTR Time (s)	Primitives	Device Tiles	WL	CPD (ns)
attention_layer	64 (1.82×)	1,013 (0.92×)	36,457 (0.78×)	21,904 (1.0×)	544,351 (0.9×)	8.8 (0.7×)
bnn	170 (1.17×)	979 (0.86×)	154,384 (0.86×)	7,569 (1.1×)	1,246,157 (1.05×)	8.6 (1.07×)
bwave_like.fixed.large	249 (▽)	1,509 (▽)	44,026 (▽)	38,416 (▽)	2,038,544 (▽)	11.1 (▽)
bwave_like.fixed.small	33 (▽)	535 (▽)	12,990 (▽)	9,604 (▽)	423,448 (▽)	9.2 (▽)
bwave_like.float.large	7,119 (▽)	16,688 (▽)	236,120 (▽)	102,400 (▽)	5,821,981 (▽)	12.1 (▽)
bwave_like.float.small	562 (▽)	1,624 (▽)	64,124 (▽)	10,816 (▽)	1,124,557 (▽)	8 (▽)
clstm_like.large	29,877 (0.61×)	34,191 (0.62×)	803,453 (0.87×)	40,000 (0.65×)	4,567,072 (0.79×)	8.7 (0.83×)
clstm_like.medium	14,193 (0.65×)	16,932 (0.65×)	538,767 (0.85×)	28,224 (0.67×)	2,788,060 (0.78×)	7.7 (0.89×)
clstm_like.small	3,239 (0.62×)	4,442 (0.63×)	274,104 (0.81×)	14,400 (0.62×)	1,255,624 (0.75×)	7.1 (0.85×)
conv_layer	107 (0.42×)	214 (0.55×)	28,067 (0.92×)	3,136 (1.0×)	234,630 (0.96×)	2 (0.94×)
conv_layer_hls	47 (1.16×)	289 (0.38×)	14,381 (1.0×)	10,816 (0.96×)	94,405 (1.03×)	5.8 (0.94×)
dla_like.large	35,627 (0.62×)	51,907 (0.72×)	967,150 (0.9×)	92,416 (0.91×)	9,639,251 (0.88×)	8.6 (0.96×)
dla_like.medium	4,976 (0.56×)	9,873 (0.74×)	332,889 (0.86×)	23,104 (0.88×)	2,632,449 (0.92×)	7.9 (1.33×)
dla_like.small	1,013 (0.56×)	3,123 (0.87×)	139,473 (0.84×)	7,744 (0.96×)	871,178 (0.91×)	5.9 (1.08×)
dnnweaver	1,034 (▽)	5,149 (▽)	128,693 (▽)	37,636 (▽)	2,699,139 (▽)	12.9 (▽)
eltwise_layer	14 (0.31×)	90 (0.68×)	12,199 (1.02×)	3,136 (1.0×)	191,778 (0.99×)	1.8 (0.98×)
gemm_layer	524 (0.74×)	849 (0.82×)	53,584 (1.01×)	13,924 (1.0×)	920,994 (1.21×)	5.1 (0.85×)
lenet	2,571 (▽)	2,716 (▽)	23,218 (▽)	1,600 (▽)	220,552 (▽)	9.6 (▽)
lstm	525 (0.04×)	3,438 (0.22×)	171,173 (0.84×)	225,00 (0.53×)	1,701,414 (0.81×)	8.9 (0.97×)
proxy.1	1,341 (▽)	10,412 (▽)	208,677 (▽)	69,696 (▽)	4,103,559 (▽)	9.2 (▽)
proxy.2	10,752 (▽)	18,210 (▽)	327,941 (▽)	35,344 (▽)	3,418,297 (▽)	8.4 (▽)
proxy.3	4,646 (▽)	6,846 (▽)	258,328 (▽)	26,896 (▽)	2,442,888 (▽)	11.6 (▽)
proxy.4	4,405 (▽)	20,671 (▽)	285,267 (▽)	49,284 (▽)	4,486,209 (▽)	10.7 (▽)
proxy.5	1,584 (▽)	3,271 (▽)	107,988 (▽)	18,496 (▽)	1,176,576 (▽)	9.8 (▽)
proxy.6	638 (▽)	8,022 (▽)	138,484 (▽)	19,044 (▽)	1,723,354 (▽)	7.6 (▽)
proxy.7	2,260 (▽)	5,758 (▽)	176,403 (▽)	19,044 (▽)	2,027,499 (▽)	9.1 (▽)
proxy.8	1,837 (▽)	3,378 (▽)	107,878 (▽)	23,104 (▽)	1,182,351 (▽)	9.1 (▽)
reduction_layer	34 (1.51×)	104 (1.02×)	14,146 (0.89×)	1,444 (1.0×)	170,821 (0.92×)	7.2 (1.03×)
robot_rl	34 (0.86×)	143 (0.38×)	23,015 (0.81×)	2,704 (1.0×)	173,867 (0.78×)	6.4 (0.91×)
softmax	28 (3.21×)	113 (1.08×)	11,781 (1.0×)	2,916 (0.87×)	126,129 (1.0×)	9.3 (1.06×)
spmv	32 (1.39×)	171 (0.81×)	14,407 (1.0×)	7,056 (1.0×)	219,926 (1.03×)	5.9 (0.97×)
tdarknet_like.large	16,638 (0.4×)	23,591 (0.44×)	278,862 (0.82×)	108,900 (0.79×)	4,115,729 (1.01×)	12 (0.7×)
tdarknet_like.small	2,936 (0.48×)	21,164 (1.28×)	127,288 (0.92×)	285,156 (2.25×)	4,110,706 (1.51×)	20.2 (1.25×)
tpu_like.large.os	1,590 (0.81×)	3,257 (0.45×)	60,173 (0.98×)	69,696 (1.0×)	2,036,045 (0.95×)	2.7 (0.99×)
tpu_like.large.ws	875 (0.7×)	2,607 (0.31×)	57,273 (0.77×)	69,696 (1.0×)	962,221 (0.96×)	3 (1.0×)
tpu_like.small.os	112 (0.64×)	459 (0.55×)	18,471 (0.97×)	18,496 (1.0×)	391,873 (0.97×)	2.4 (1.02×)
tpu_like.small.ws	107 (0.62×)	505 (0.52×)	20,285 (0.82×)	18,496 (1.0×)	265,961 (0.96×)	2.9 (0.95×)
Geomean	596 (0.67×)	2,374 (0.62×)	83,514 (0.89×)	18,499 (0.93×)	1,068,031 (0.95×)	7.1 (0.96×)

▽: The design failed synthesis with Odin-II.

to an overall VTR 9 runtime of 55% of the VTR 8 runtime while achieving slightly improved quality (4% less WL and almost the same CPD). VTR 9 increases memory footprint by only 4% compared to VTR 8 despite its more flexible 3D architecture model. Overall VTR 9 has equivalent performance to VTR 8 on smaller designs (as indicated by the VTR benchmarks) and significantly faster runtimes on larger circuits (as shown by Titan results).

10.3 Parallel Results

The results presented in the previous subsection compare the single-threaded mode of both VTR 8 and VTR 9 (i.e., running on one CPU core). However, VTR 8 can parallelize timing analysis across multiple threads, while VTR 9 can parallelize both timing analysis and routing, as described in Section 8.8.

Figure 20 shows the geometric average results over the Titan23 benchmarks, normalized to the results of single-threaded VTR 8. VTR 8 and 9 have similar CPD values in this test, but VTR 9

Table 16. VTR 9 QoR for the VTR Benchmarks (Normalized to VTR 8 between Brackets) Averaged over Three Different Seeds

Circuit	Synth Time (s)	Pack Time (s)	Place Time (s)	Route Time (s)	Flow Time (s)	Peak Mem (GB)	Primitives	Tiles	WL	CPD (ns)	Min. W
`LU32PEEng`	1,081 (1.06×)	271 (0.82×)	563 (0.91×)	52 (0.38×)	2,276 (1.0×)	2.13 (1.04×)	108,159 (1.07×)	10,609 (1.08×)	1,344,088 (1.01×)	73.4 (0.96×)	123 (0.89×)
`LU8PEEng`	154 (1.5×)	78 (0.8×)	78 (0.76×)	14 (0.79×)	404 (1.1×)	0.61 (1.22×)	31,925 (1.02×)	3,136 (1.04×)	329,864 (1.01×)	75.6 (0.97×)	93 (1.05×)
`bgm`	133 (0.58×)	94 (1.72×)	100 (1.09×)	17 (1.4×)	442 (1.01×)	0.7 (1.51×)	33,722 (1.36×)	3,969 (1.27×)	386,083 (1.31×)	19.3 (0.98×)	72 (0.99×)
`mcml`	3,063 (0.89×)	252 (0.85×)	656 (0.77×)	42 (0.54×)	4,303 (0.88×)	2.24 (1.0×)	159,806 (0.96×)	9,604 (0.94×)	948,978 (0.92×)	46.7 (1.05×)	131 (0.98×)
`stereovision0`	21 (1.46×)	11 (1.09×)	12 (0.71×)	3 (1.33×)	66 (0.89×)	0.23 (1.14×)	21,365 (0.98×)	1,156 (1.06×)	58,998 (0.91×)	3.7 (1.05×)	48 (0.88×)
`stereovision1`	22 (0.41×)	11 (1.14×)	16 (0.75×)	4 (0.78×)	77 (0.66×)	0.29 (1.15×)	19,301 (0.99×)	1,600 (1.0×)	124,936 (0.96×)	5.4 (0.99×)	80 (0.99×)
`stereovision2`	23 (2.01×)	31 (0.85×)	71 (0.49×)	12 (0.82×)	265 (0.97×)	1.02 (1.13×)	37,074 (0.88×)	6,400 (0.87×)	386,317 (0.89×)	14.1 (0.98×)	93 (0.99×)
Geomean	118 (0.91×)	57 (1.0×)	87 (0.76×)	13 (0.79×)	454 (0.93×)	0.75 (1.16×)	43,281 (1.03×)	3,950 (1.03×)	333,884 (0.99×)	20.3 (1.0×)	87 (0.96×)

Table 17. VTR 9 QoR for the Titan Benchmarks (Normalized to VTR 8 between Brackets) Averaged over Three Different Seeds

Circuit	Synth Time (s)	Pack Time (s)	Place Time (s)	Route Time (s)	Flow Time (s)	Peak Mem (GB)	Primitives	Tiles	WL	CPD (ns)	Min. W
`LU230`	-	1,554 (0.76×)	4,172 (0.38×)	696 (0.04×)	8,367 (0.25×)	19.49 (1.05×)	568,001 (1.0×)	137,170 (1.0×)	16,669,336 (1.04×)	24 (0.96×)	-
`LU_Network`	-	1,036 (0.99×)	5,330 (0.63×)	401 (0.77×)	7,501 (0.68×)	11.23 (1.03×)	630,079 (1.0×)	358,60 (1.0×)	6,274,492 (1.06×)	9.4 (0.99×)	-
`SLAM_spheric`	-	257 (1.0×)	588 (0.65×)	145 (0.96×)	1,122 (0.75×)	2.82 (1.07×)	111,354 (1.0×)	6,650 (0.95×)	1,643,361 (1.08×)	80.6 (0.99×)	-
`bitcoin_miner`	-	1,467 (1.45×)	10,993 (0.57×)	550 (0.51×)	13,892 (0.61×)	14.34 (1.13×)	1,087,537 (1.0×)	41,300 (1.1×)	9,543,478 (0.86×)	8.8 (1.12×)	-
`bitonic_mesh`	-	688 (0.96×)	1,348 (0.36×)	306 (0.65×)	2,957 (0.51×)	7.02 (0.99×)	190,746 (1.0×)	43,318 (1.0×)	4,326,100 (0.94×)	14.1 (0.98×)	-
`cholesky_bdti`	-	387 (1.01×)	1,080 (0.52×)	266 (0.48×)	2,087 (0.6×)	5.43 (0.99×)	255,478 (1.0×)	21,125 (1.0×)	2,650,299 (1.0×)	9.8 (1.08×)	-
`cholesky_mc`	-	159 (1.04×)	340 (0.49×)	110 (0.63×)	817 (0.6×)	3.09 (1.01×)	108,592 (1.0×)	11,625 (1.0×)	1,124,780 (0.89×)	7.4 (1.02×)	-
`dart`	-	543 (1.1×)	776 (0.58×)	141 (0.87×)	1,715 (0.74×)	4.14 (1.06×)	202,401 (1.0×)	14,076 (1.0×)	2,069,734 (0.9×)	14.5 (0.96×)	-
`denoise`	-	532 (0.97×)	3,161 (0.64×)	336 (0.96×)	4,374 (0.69×)	5.9 (1.07×)	274,786 (1.0×)	16,390 (0.97×)	3,061,392 (0.94×)	868.1 (1.01×)	-
`des90`	-	385 (1.01×)	609 (0.43×)	158 (0.7×)	1,466 (0.59×)	4.08 (1.0×)	110,549 (1.0×)	21,717 (1.0×)	2,191,757 (0.93×)	13.2 (1.02×)	-
`directrf`	-	1,538 (0.87×)	11,813 (0.58×)	692 (0.04×)	15,429 (0.37×)	19.5 (1.06×)	930,989 (1.0×)	74,495 (1.0×)	11,850,872 (0.86×)	10.6 (0.82×)	-
`gsm_switch`	-	998 (0.96×)	2,470 (0.45×)	242 (0.19×)	4,466 (0.51×)	9.52 (1.01×)	490,068 (1.0×)	48,195 (1.0×)	5,290,385 (0.98×)	9.9 (1.18×)	-
`mes_noc`	-	1,996 (1.03×)	4,042 (0.59×)	439 (0.86×)	7,036 (0.7×)	9.09 (1.05×)	547,568 (1.0×)	27,936 (0.99×)	5,060,432 (1.0×)	11.8 (0.95×)	-
`minres`	-	526 (1.04×)	1,055 (0.36×)	183 (0.68×)	2,298 (0.51×)	6.73 (1.0×)	257,480 (1.0×)	37,575 (1.01×)	2,782,423 (0.98×)	8.5 (1.05×)	-
`neuron`	-	114 (1.0×)	218 (0.39×)	64 (0.96×)	583 (0.54×)	2.85 (1.01×)	86,875 (1.0×)	12,384 (1.0×)	752,138 (0.98×)	8.3 (1.01×)	-
`openCV`	-	461 (1.03×)	1,002 (0.45×)	234 (0.38×)	2,175 (0.55×)	5.85 (0.97×)	212,552 (1.0×)	32,395 (1.0×)	3,233,360 (0.95×)	10.4 (0.96×)	-
`segmentation`	-	258 (1.0×)	1,089 (0.65×)	183 (0.99×)	1,772 (0.72×)	3.62 (1.05×)	137,832 (1.0×)	13,736 (1.0×)	1,676,173 (0.95×)	852.1 (1.0×)	-
`sparcT1_chip2`	-	2,219 (1.0×)	6,000 (0.6×)	562 (0.76×)	9,827 (0.2×)	12.67 (1.06×)	760,412 (1.0×)	57,960 (1.0×)	7,529,913 (0.94×)	19.4 (0.87×)	-
`sparcT1_core`	-	337 (1.01×)	275 (0.58×)	102 (1.09×)	821 (0.7×)	2.33 (1.15×)	91,975 (1.0×)	5,002 (1.0×)	1,263,420 (1.03×)	9.3 (1.11×)	-
`sparcT2_core`	-	1,101 (1.1×)	2,085 (0.63×)	306 (0.45×)	3,834 (0.71×)	5.51 (1.09×)	300,220 (1.0×)	17,556 (1.02×)	4,704,034 (1.02×)	11.3 (1.09×)	-
`stap_qrd`	-	381 (1.07×)	1,709 (0.63×)	166 (0.57×)	2,573 (0.68×)	4.89 (1.03×)	234,177 (1.0×)	18,486 (0.99×)	2,610,952 (0.94×)	8 (1.17×)	-
`stereo_vision`	-	98 (0.93×)	168 (0.3×)	38 (0.8×)	481 (0.44×)	2.83 (1.12×)	94,090 (1.0×)	12,384 (1.0×)	586,325 (0.92×)	7.6 (0.87×)	-
Geomean	-	550 (1.01×)	1,441 (0.51×)	226 (0.51×)	2,765 (0.55×)	6.05 (1.04×)	257,271 (1.0×)	23,692 (1.0×)	3,116,242 (0.96×)	17.6 (1.0×)	-

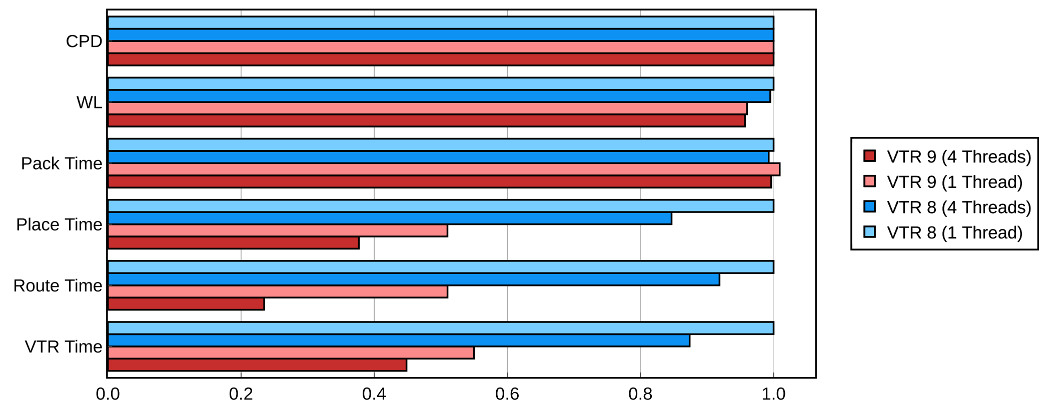


Fig. 20. QoR comparison of VTR 9 and VTR 8 using one and four threads on the Titan23 benchmarks. All results are normalized to the single-threaded VTR 8 and averaged over three placement seeds.

reduces WL by 4% vs. VTR 8. QoR is not degraded by parallelism in either version, as WL and CPD are essentially the same as thread count changes. Enabling parallel timing analysis reduces VTR 8's runtime by 13%, while enabling parallel timing analysis and (baseline) parallel routing reduces VTR 9's overall runtime by 18%. Overall, VTR 9 has significantly reduced the runtime of different physical implementation stages; the serial VTR 9 algorithms reduce runtime by 45% vs. serial VTR 8, while in parallel mode VTR 9 has 49% less runtime than VTR 8. The gains are largest on the two most CPU-intensive algorithms: the VTR 9 baseline parallel router and placement engine with parallel timing analysis are 4× and 2.2× faster than the VTR 8 equivalents run in parallel mode, respectively.

11 Conclusion

The VTR 9 release enhances this open source CAD tool suite in several major ways. The new Parmys synthesis engine combines the strong language coverage of Yosys with architecture-aware hard block optimizations. This not only improves synthesis quality but also enables VTR to implement a wider variety of benchmarks that make use of advanced Verilog constructs. The VTR physical implementation stages (i.e., packing, placement, and routing) have also been enhanced in several ways. The placement engine now features a smarter initial placement and a variety of targeted placement perturbations dynamically selected by an RL agent. The VTR router has been enhanced so that it can perform intra- and inter-cluster routing in a single (flat) stage, improving its ability to optimize for recent FPGAs with less flexible local interconnect. The routing algorithm has also been made parallel through a spatial partitioning and net decomposition approach; this not only reduces routing time by 2.4× but also makes a well tested parallel routing infrastructure available to the community for future algorithmic experimentation. VTR now includes flexible floorplanning constraints that both the packer and placer respect; this allows end users to control design implementation details and to experiment with divide-and-conquer CAD flows that could reduce compile time. Overall, these major enhancements reduce the flow's runtime by 48% (a 1.93× speed up) on average across the Titan23 benchmark suite, with equal or better CPD and WL optimization.

Additionally, the VTR architecture description language has been extended to model more general programmable routing architectures with varying channel widths and complex (e.g., L-shaped, T-shaped) wires. These enhancements enable architecture captures of the Intel/Altera Stratix 10 and AMD/Xilinx 7-series FPGAs, which are included in this release so researchers can use them as a baseline onto which new architecture ideas can be overlaid. VTR 9 can now model and optimize for multi-layer 3D-stacked FPGAs and FPGAs with embedded NoCs, enabling a broad range of new architecture investigations.

12 Future Work

There are many directions for future extensions of VTR. One is to continue to improve the CAD algorithms to increase result quality, reduce runtime and add additional automation to keep designers productive despite ever-larger FPGA designs. Recent work has shown that generating a primitive-level analytic placement [113] that is then refined by VTR's annealer leads to a better solution than either analytic techniques or annealing alone [92]. However, the work in [92] has limited architectural flexibility; integrating a mixed analytic/annealing placement engine that is fully data-driven and can target the wide range of VTR architectures would benefit the community. Another direction to improve runtime is to leverage parallelism; VTR 9 already contains parallel routers and a parallel timing analysis engine, but other CPU-intensive phases such as logic synthesis, packing, and placement could be also made parallel. A third approach is to extend the floorplanning constraints discussed in Section 8.5 to also control routing, enabling research into CAD flows and architectures that facilitate the stitching of pre-placed and routed design components. The floorplanning constraints in VTR 9 are *mandatory*; they are always obeyed by the placement engine. Future research could investigate the utility of also allowing *hints* by designers, for example by using floorplanning constraints to guide packing and the early stages of placement, but allowing them to be violated later in the anneal if it would improve timing or WL. Currently to use NoCs in the VTR flow a designer must instantiate NoC access points (logical routers) in the design, thereby specifying which communication occurs on the NoC. Future work could explore automatically moving latency-insensitive communication to a NoC when it is beneficial, and leaving it in the programmable fabric when that provides a better match to that traffic flow's latency and bandwidth needs.

A second large area for research is to explore the very wide range of architectures VTR 9 now supports. While prior research [114, 115] has shown the value of hard NoCs on FPGAs, much work remains to find the best NoC topologies and NoC link bandwidths to support the high traffic flow bandwidths, particularly near I/O interfaces, in recent FPGA designs. The NoC-aware algorithms in VTR 9 and the NoC benchmarks in the Hermes suite enable work in this direction. As Moore's law has slowed down, systems have moved more functionality to hardened components and have turned to die stacking to overcome I/O bottlenecks. The 3D architecture capabilities of VTR 9 allow FPGA architects to explore die stacking ideas, while its support for embedded NoCs enables investigations into FPGAs with NoC-attached domain-specific accelerators [116]. Finally, scaling the FPGA fabric itself faces major challenges; shrinking interconnect increases resistance, and the SRAM cells heavily used in FPGA LUTs and programmable routing muxes are not scaling well. VTR 9 has the capabilities needed by architects to explore new interconnect and logic structures that better suit next-generation manufacturing technologies.

As VTR has become the implementation CAD tool for some commercial and several fabricated academic FPGAs, a third future direction is to add utilities to the VTR flow that benefit those implementing FPGA designs, as well as those researching new FPGAs. The **Interactive Path Analysis (IPA)** tool developed by QuickLogic[3] creates a user-friendly interactive timing path viewer that maintains a live connection to VTR's timing analyzer and router to extract and view timing paths. The FASM tool developed by Google enables creation of a programming bitstream

[3]The IPA tool is available at https://github.com/w0lek/IPAClient.

for novel FPGAs. We believe there are opportunities to create many other utilities to ease design creation, floorplanning, and analysis in the VTR ecosystem.

This release of VTR combines the work of many research groups around the world, and we look forward to seeing the new investigations enabled by its features.

Acknowledgements

The authors would like to thank all those who have contributed to this release of the VTR project, including Arash Ahmadian, Tim Ansell, Amir Arjomandaghaie, Aman Arora, Sebastian Arzayus, Alireza Azadi, David Baines, Alan Bu, Alessandro Comodi, Helen Dai, Seyed Damghani, Dusty DeWeese, Mohamed Eldafrawy, Mahshad Farahani, Peter Grossman, Karol Gugala, Dana How, Bill Hu, Navid Jafarof, Sara Khalid, Tao Li, Yulang Luo, Mahesh Madhav, Jennifer Mah, Meet Patel, Paula Perdomo, Aleksandr Pivovarov, Keith Rothman, Nathan Shreve, Alex Singer, Srivatsan Srinivasan, Xifan Tang, Hang Yan, Tina Yang, and Henner Zeller. We also thank Mark Bourgeault, Mahesh Iyer, and Jason Luu for insightful discussions.

References

[1] Gordon E. Moore. 1998. Cramming more components onto integrated circuits. *Proceedings of the IEEE* 86, 1 (1998), 82–85.

[2] John Shalf. 2020. The future of computing beyond Moore's law. *Philosophical Transactions of the Royal Society A* 378 (2020), 2166.

[3] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. 2021. Ten lessons from three generations shaped Google's TPUv4i: Industrial product. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[4] Michael Anderson, Benny Chen, Stephen Chen, Summer Deng, Jordan Fix, Michael Gschwind, Aravind Kalaiah, Changkyu Kim, Jaewon Lee, Jason Liang, et al. 2021. First-generation inference accelerator deployment at Facebook. arXiv:2107.04140. Retrieved from https://arxiv.org/abs/2107.04140

[5] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[6] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[7] Jordane Lorandel, Jean-Christophe Prevotet, and Maryline Helard. 2016. Fast power and performance evaluation of FPGA-based wireless communication systems. *IEEE Access* 4 (2016), 2005–2018.

[8] Rui Ma, Evangelos Georganas, Alexander Heinecke, Sergey Gribok, Andrew Boutros, and Eriko Nurvitadhi. 2022. FPGA-based AI smart NICs for scalable distributed AI training systems. *IEEE Computer Architecture Letters* 21, 2 (2022), 49–52.

[9] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps intrusion prevention on a single server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[11] Yi-Chieh Kao, Hung-An Chen, and Hsi-Pin Ma. 2022. An FPGA-based high-frequency trading system for 10 gigabit ethernet with a latency of 433 ns. In *IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*.

[12] Mathew Hall and Vaughn Betz. 2020. From TensorFlow graphs to LUTs and wires: Automated sparse and physically aware CNN hardware generation. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[13] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[14] Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, et al. 2019. Why compete when you can work together: FPGA-ASIC integration for persistent RNNs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[15] Ho-Cheung Ng, Shuanglong Liu, and Wayne Luk. 2017. Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[16] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. 2020. Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[17] Andrew Boutros and Vaughn Betz. 2021. FPGA architecture: Principles and progression. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 4–29.

[18] Advanced Micro Devices, Inc. 2024. *AMD Versal Premium Series*. Product Selection Guide (XMP463).

[19] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. 2021. Stratix 10 NX architecture and applications. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[20] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. 2017. Don't forget the memory: Automatic block RAM modelling, optimization, and architecture exploration. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[21] Aman Arora, Andrew Boutros, Daniel Rauch, Aishwarya Rajen, Aatman Borda, Seyed Alireza Damghani, Samidh Mehta, Sangram Kate, Pragnesh Patel, Kenneth B. Kent, et al. 2021. Koios: A deep learning benchmark suite for FPGA architecture and CAD research. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

[22] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2018. Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[23] Aman Arora, Moinak Ghosh, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2022. Tensor slices: FPGA building blocks for the deep learning era. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (2022), 1–34.

[24] Grace Zgheib, Liqun Yang, Zhihong Huang, David Novo, Hadi Parandeh-Afshar, Haigang Yang, and Paolo Ienne. 2014. Revisiting and-inverter cones. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[25] Mohamed Eldafrawy, Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2020. FPGA logic block architectures for efficient deep learning inference. *ACM Transactions on Reconfigurable Technology and Systems* 13, 3 (2020), 1–34.

[26] Charles Chiasson and Vaughn Betz. 2013. Should FPGAs abandon the pass-gate? In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[27] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. 2012. The VTR project: Architecture and CAD for FPGAs from Verilog to routing. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

[28] Jason Luu, Jonathan Rose, and Jason Anderson. 2014. Towards interconnect-adaptive packing for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[29] 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, et al. 2020. VTR 8: High-performance CAD and customizable FPGA architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems* 13, 2 (2020), 1–55.

[30] Srivatsan Srinivasan, Andrew Boutros, Fatemehsadat Mahmoudi, and Vaughn Betz. 2023. Placement optimization for NoC-enhanced FPGAs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[31] Andrew Boutros, Fatemehsadat Mahmoudi, Amin Mohaghegh, Stephen More, and Vaughn Betz. 2023. Into the third dimension: Architecture exploration tools for 3D reconfigurable acceleration devices. In *IEEE International Conference on Field Programmable Technology (FPT)*.

[32] Amin Mohaghegh and Vaughn Betz. 2023. Tear down the wall: Unified and efficient intra- and inter-cluster routing for FPGAs. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

[33] Kimia Talaei Khoozani, Arash Ahmadian Dehkordi, and Vaughn Betz. 2023. Titan 2.0: Enabling open-source CAD evaluation with a modern architecture capture. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

[34] Daniel Khadivi. 2023. *Parmys: Odin-II Intelligent Partial Mapper for Yosys Synthesis Suite*. Master of Computer Science Thesis.

[35] Aman Arora, Andrew Boutros, Seyed Alireza Damghani, Karan Mathur, Vedant Mohanty, Tanmay Anand, Mohamed A. Elgammal, Kenneth B. Kent, Vaughn Betz, Lizy K. John, et al. 2023. Koios 2.0: Open-source deep learning benchmarks for FPGA architecture and CAD research. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 11 (2023), 3895–3909.

[36] Fatemehsadat Mahmoudi, Mohamed A. Elgammal, Soheil Gholami Shahrouz, Kevin E. Murray, and Vaughn Betz. 2023. Respect the difference: Reinforcement learning for heterogeneous FPGA placement. In *IEEE International Conference on Field Programmable Technology (FPT)*.

[37] Clifford Wolf. 2016. Yosys Open Synthesis Suite. Retrieved from https://github.com/YosysHQ/yosys

[38] Christian Beckhoff, Dirk Koch, and Jim Torresen. 2011. The Xilinx Design Language (XDL): Tutorial and use cases. In *IEEE International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*.

[39] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. 2011. Torc: Towards an open-source tool flow. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[40] Travis Haroldsen, Brent Nelson, and Brad Hutchings. 2015. RapidSmith 2: A framework for BEL-level CAD exploration on Xilinx FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[41] Chris Lavin and Alireza Kaviani. 2018. RapidWright: Enabling custom crafted implementations for FPGAs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[42] Shawn Malhotra, Terry P. Borer, Deshanand P. Singh, and Stephen Dean Brown. 2004. The Quartus University Interface Program: Enabling advanced FPGA research. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[43] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, et al. 2003. The Stratix II routing and logic architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[44] Akshay Sharma, Scott Hauck, and Carl Ebeling. 2005. Architecture-adaptive routability-driven placement for FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[45] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+nextpnr: An open source framework from Verilog to bitstream for commercial FPGAs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[46] Aman Arora, Atharva Bhamburkar, Aatman Borda, Tanmay Anand, Rishabh Sehgal, Bagus Hanindhito, Pierre-Emmanuel Gaillardon, Jaydeep Kulkarni, and Lizy K. John. 2023. CoMeFa: Deploying compute-in-memory on FPGAs for deep learning acceleration. *ACM Transactions on Reconfigurable Technology and Systems* 16, 3 (2023), 1–34.

[47] Stefan Nikolić, Grace Zgheib, and Paolo Ienne. 2020. Straight to the point: Intra-and intercluster LUT connections to mitigate the delay of programmable routing. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[48] Stefan Nikolić and Paolo Ienne. 2021. Turning PathFinder upside-down: Exploring FPGA switch-blocks by negotiating switch presence. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

[49] Fatemeh Serajeh Hassani, Mohammad Sadrosadati, Nezam Rohbani, Sebastian Pointner, Robert Wille, and Hamid Sarbazi-Azad. 2024. An efficient FPGA architecture with turn-restricted switch boxes. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–18.

[50] Kaichuang Shi, Xuegong Zhou, Hao Zhou, and Lingli Wang. 2022. An optimized GIB routing architecture with bent wires for FPGA. *ACM Transactions on Reconfigurable Technology and Systems* 16, 1 (2022), 1–28.

[51] Mahdi Abbaszadeh and Dana L. How. 2024. From topology to realization in FPGA/VPR routing. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

[52] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. 2017. 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 (2017), 629–642.

[53] Mohamed A. Elgammal and Vaughn Betz. 2022. Quality & generality: A flexible FPGA re-clustering technique to improve packing and placement. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[54] Andrew David Gunter and Steven Wilton. 2023. Towards a machine learning approach to predicting the difficulty of FPGA routing problems. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

[55] Ruichen Chen, Shengyao Lu, Mohamed A. Elgammal, Peter Chun, Vaughn Betz, and Di Niu. 2023. VPR-Gym: A platform for exploring AI techniques in FPGA placement optimization. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

[56] Sandeep Sunkavilli, Zhiming Zhang, and Qiaoyan Yu. 2021. Analysis of attack surfaces and practical attack examples in open-source FPGA CAD tools. In *IEEE International Symposium on Quality Electronic Design (ISQED)*.

[57] Sandeep Sunkavilli, Zhiming Zhang, and Qiaoyan Yu. 2021. New security threats on FPGAs: From FPGA design tools perspective. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[58] Kevin E. Murray, Mohamed A. Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. 2020. SymbiFlow and VPR: An open-source design flow for commercial and novel FPGAs. *IEEE Micro* 40, 4 (2020), 49–57.

[59] Xifan Tang, Ganesh Gore, Edouard Giacomin, Aurélien Alacchi, Baudouin Chauviere, and Pierre-Emmanuel Gaillardon. 2020. OpenFPGA: Towards automated prototyping for versatile FPGAs. In *Workshop on Open-Source EDA Technology*.

[60] Jin Hee Kim and Jason H. Anderson. 2017. Synthesizable standard cell FPGA fabrics targetable by the Verilog-to-Routing CAD flow. *ACM Transactions on Reconfigurable Technology and Systems* 10, 2 (2017), 1–23.

[61] Brett Grady and Jason H. Anderson. 2018. Synthesizable heterogeneous FPGA fabrics. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[62] Prashanth Mohan, Oguz Atli, Onur Kibar, Mohammed Zackriya, Larry Pileggi, and Ken Mai. 2021. Top-down physical design of soft embedded FPGA fabrics. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[63] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. 2021. FABulous: An embedded FPGA framework. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[64] Ang Li and David Wentzlaff. 2021. PRGA: An open-source FPGA research and prototyping framework. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[65] Kevin E. Murray, Sheng Zhong, and Vaughn Betz. 2020. AIR: A fast but lazy timing-driven FPGA router. In *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[66] The Verilog-to-Routing Project. 2025. FPGA Architecture Description. Retrieved from https://docs.verilogtorouting.org/en/latest/arch/#fpga-architecture-description

[67] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. 2019. Network-on-chip programmable platform in Versal^{TM} ACAP architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[68] Achronix Semiconductor Corp. 2019. *Speedster7t Network on Chip*. User Guide (UG089).

[69] Supriya Velagapudi and Mark Honman. 2022. *Addressing Memory-Bandwidth and Compute-Intensive Challenges with Intel Agilex M-Series FPGAs*. Whitepaper, WP-01313-1.0.

[70] Wilfred Gomes, Slade Morgan, Boyd Phelps, Tim Wilson, and Erik Hallnor. 2022. Meteor lake and arrow lake Intel next-gen 3D client architecture platform with Foveros. In *IEEE Hot Chips Symposium (HCS)*.

[71] Ming-Fa Chen, Fang-Cheng Chen, Wen-Chih Chiou, and Doug C. H. Yu. 2019. System on Integrated Chips (SoIC^{TM}) for 3D heterogeneous integration. In *IEEE Electronic Components and Technology Conference (ECTC)*.

[72] John Wuu, Rahul Agarwal, Michael Ciraula, Carl Dietz, Brett Johnson, Dave Johnson, Russell Schreiber, Raja Swaminathan, Will Walker, and Samuel Naffziger. 2022. 3D V-Cache: The implementation of a hybrid-bonded 64MB stacked cache for a 7nm x86-64 CPU. In *IEEE International Solid-State Circuits Conference (ISSCC)*.

[73] Chirag Ravishankar, Dinesh Gaitonde, and Trevor Bauer. 2018. Placement strategies for 2.5D FPGA fabric architectures. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[74] Cristinel Ababei, Yan Feng, Brent Goplen, Hushrav Mogal, Tianpei Zhang, Kia Bazargan, and Sachin Sapatnekar. 2005. Placement and routing in 3D integrated circuits. *IEEE Design & Test of Computers* 22, 6 (2005), 520–531.

[75] Vaughn Betz and Jonathan Rose. 1996. Directional bias and non-uniformity in FPGA global routing architectures. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

[76] Satish Sivaswamy, Gang Wang, Cristinel Ababei, Kia Bazargan, Ryan Kastner, and Eli Bozorgzadeh. 2005. HARP: Hard-wired routing pattern FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[77] Morten B. Petersen, Stefan Nikolić, and Mirjana Stojilović. 2021. NetCracker: A peek into the routing architecture of Xilinx 7-series FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[78] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2013. Titan: Enabling large and complex benchmarks in academic CAD. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*.

[79] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[81] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556. Retrieved from https://arxiv.org/abs/1409.1556

[82] Anupreetham Anupreetham, Mohamed Ibrahim, Mathew Hall, Andrew Boutros, Ajay Kuzhively, Abinash Mohanty, Eriko Nurvitadhi, Vaughn Betz, Yu Cao, Jae-Sun Seo, et al. 2024. High throughput FPGA-based object detection via algorithm-hardware co-design. *ACM Transactions on Reconfigurable Technology and Systems* 17, 1 (2024), 1–20.

[83] Soheil Gholami Shahrouz and Vaughn Betz. 2024. The road less traveled: Congestion-aware NoC placement and packet routing for FPGAs. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

[84] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. 2016. The Stratix^{TM} 10 highly pipelined FPGA architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[85] Xilinx, Inc. 2016. *7 Series FPGAs Configurable Logic Block*. User Guide (UG474 v1.8).

[86] Eddie Hung, Fatemeh Eslami, and Steven J. E. Wilton. 2013. Escaping the academic sandbox: Realizing VPR circuits on Xilinx devices. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[87] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: Versal$^{\text{TM}}$ architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[88] F4PGA. 2017. Project X-Ray: Documenting the Xilinx 7-Series Bitstream Format. Retrieved from https://github.com/f4pga/prjxray

[89] Georgiy Krylov, Jean-Philippe Legault, and Kenneth B. Kent. 2020. Hard and soft logic trade-offs for multipliers in VTR. In *IEEE EuroMICRO Conference on Digital System Design (DSD)*.

[90] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *Springer International Conference on Computer Aided Verification (CAV)*.

[91] Jan Rychlewski. 1984. On Hooke's law. *Journal of Applied Mathematics and Mechanics* 48, 3 (1984), 303–314.

[92] Rachel Rajarathnam, Kate Thurmer, Vaughn Betz, Mahesh A. Iyer, and David Z. Pan. 2024. Better together: Combining analytical and annealing methods for FPGA placement. In *IEEE International Conference on Field Programmable Logic (FPL)*.

[93] Mohamed A. Elgammal, Kevin E. Murray, and Vaughn Betz. 2022. RLPlace: Using reinforcement learning and smart perturbations to optimize FPGA placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 8 (2021), 2532–2545.

[94] Gang Chen and Jason Cong. 2005. Simultaneous timing-driven placement and duplication. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[95] Mohamed A. Elgammal, Kevin E. Murray, and Vaughn Betz. 2020. Learn to place: FPGA placement using reinforcement learning and directed moves. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[96] Christopher J. Glass and Lionel M. Ni. 1992. The turn model for adaptive routing. *ACM SIGARCH Computer Architecture News* 20, 2 (1992), 278–287.

[97] Ge-Ming Chiu. 2000. The odd-even turn model for adaptive routing. *IEEE Transactions on Parallel and Distributed Systems* 11, 7 (2000), 729–738.

[98] Douglas C. H. Yu, Chuei-Tang Wang, and Harry Hsia. 2021. Foundry perspectives on 2.5D/3D integration and roadmap. In *IEEE International Electron Devices Meeting (IEDM)*.

[99] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A negotiation-based performance-driven router for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[100] Stefan Nikolić, Francky Catthoor, Zsolt Tőkei, and Paolo Ienne. 2021. Global is the new local: FPGA architecture at 5nm and beyond. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[101] Jeffrey Chromczak, Mark Wheeler, Charles Chiasson, Dana How, Martin Langhammer, Tim Vanderhoek, Grace Zgheib, and Ilya Ganusov. 2020. Architectural enhancements in Intel Agilex FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[102] Marcel Gort and Jason H. Anderson. 2010. Deterministic multi-core parallel routing for FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[103] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[104] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[105] Fahrican Koşar, Mirjana Stojilović, and Vaughn Betz. 2024. Parallel FPGA routing with on-the-fly net decomposition. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[106] Kevin E. Murray and Vaughn Betz. 2018. Tatum: Parallel timing analysis for faster design cycles and improved optimization. In *IEEE International Conference on Field-Programmable Technology (FPT)*.

[107] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux utility for resource management. In *Springer Workshop on Job Scheduling Strategies for Parallel Processing*.

[108] Digital Research Alliance of Canada. 2024. Advanced Research Computing. Retrieved from https://alliancecan.ca/en/services/advanced-research-computing

[109] QuickLogic Corporation. 2023. Aurora 2.4 Development Tool: Celebrating the Power of Open-Source Innovation. Retrieved from https://www.quicklogic.com/2023/11/16/aurora-2-4-development-tool-celebrating-the-power-of-open-source-innovation/

[110] RapidSilicon. 2024. Rapid Silicon Leads the Way with First Complete Open-Source FPGA EDA Tool-Chain. Retrieved from https://rapidsilicon.com/rapid-silicon-leads-the-way-with-first-complete-open-source-fpga-eda-tool-chain

[111] RapidFlex. 2024. ECO System—OpenFPGA. Retrieved from https://rapid-flex.com/eco/

[112] Sadegh Yazdanshenas and Vaughn Betz. 2019. COFFE 2: Automatic modelling and optimization of complex and heterogeneous FPGA architectures. *ACM Transactions on Reconfigurable Technology and Systems* 12, 1 (2019), 1–27.

[113] Wuxi Li, Yibo Lin, and David Z. Pan. 2019. elfPlace: Electrostatics-based placement for large-scale heterogeneous FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[114] Mohamed S. Abdelfattah, Andrew Bitar, and Vaughn Betz. 2015. Take the highway: Design for embedded NoCs on FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[115] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Bala Jayadev, Jeff Cuppett, Abbas Morshed, Brian Gaide, and Ygal Arbel. 2019. Versal network-on-chip (NoC). In *IEEE Symposium on High-Performance Interconnects (HOTI)*.

[116] Andrew Boutros, Stephen More, and Vaughn Betz. 2023. A whole new world: How to architect beyond-FPGA reconfigurable acceleration devices? In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*.

Received 16 December 2024; revised 25 March 2025; accepted 28 April 2025