VIPER: A VTR Interface for Placement with Error Resilience

Kate Thurmer
kate.thurmer@mail.utoronto.ca
Electrical and Computer
Engineering Department
University of Toronto
Toronto, ON, Canada

Vaughn Betz
vaughn@eecg.utoronto.ca
Electrical and Computer
Engineering Department
University of Toronto
Toronto, ON, Canada

ABSTRACT

The open source Verilog-to-Routing (VTR) tool flow can produce legal placement solutions for arbitrarily complex FPGA architectures and is thus widely used for novel device development as well as CAD tool research. VTR's versatility is enabled by both its robust device modeling capability and its use of pre-placement clustering to abstract away complexity and maintain scalability. Clustering is not always necessary; recent academic tools demonstrate that delaying or omitting it can improve result quality for some device architectures. By incorporating a variety of external placement tools, VTR can maintain both versatility and result quality as FPGAs scale and diversify. Tool developers can benefit as well from access to VTR; however, due to VTR's complex, hierarchical device modeling, its place and route interface requires a level of detail and accuracy that is beyond the scope of many external tools. To lower the barrier to interoperability, we introduce a **V**TR **I**nterface for **P**lacement with **E**rror **R**esilience (*VIPER*). *VIPER* constructs a complete, legal, VTR-compatible clustering and placement solution based on a simplified and potentially illegal input placement.

CCS CONCEPTS

• **Hardware → Software tools for EDA**.

KEYWORDS

FPGA, Packing, Clustering, Legalization, Placement

ACM Reference Format:
Kate Thurmer and Vaughn Betz. 2024. VIPER: A VTR Interface for Placement with Error Resilience. In *14th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'24)) (HEART '24), June 19–21, 2024, Porto, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3665283.3665300

1 INTRODUCTION

As the processor speed race slows and the cost of custom ASIC development rises, FPGAs are taking on an increasingly prominent role in advanced computing, appearing in diverse application spaces including data centers [11], satellites [26], and 5G telecommunications systems [33] [2]. Domain-specific FPGAs with hardened functional units offer a cost effective alternative to dedicated ASICs (e.g. for RF signal processing [29] [18]) and have been shown to rival the performance of GPUs in artificial intelligence applications [6].

A contemporary FPGA is made up of configurable blocks of various types (e.g. DSP, RAM, IO, soft logic), each comprised of locally interconnected primitive sites (e.g. LUTs, FFs, memories, multipliers), connected by a general configurable interconnect; additional dedicated connections support multi-block structures such as carry chains [5]. Each configurable block type may offer custom local interconnect and performance-boosting functional units (e.g. adders, accumulators, domain-specific accelerators). Configurable blocks or their sub-blocks may support multiple, mutually exclusive modes of operation; for example, an IO block can operate as an inpad or an outpad, a fracturable LUT can operate as one LUT or two, and a memory block may support multiple aspect ratios [31].

Figure 1 illustrates a typical FPGA CAD flow: *synthesis* translates a high level circuit description into nets and primitives (e.g. LUTs, FFs, memories, multipliers) compatible with a target device; *placement* maps primitives to device sites; and *routing* connects nets between placed primitives using the device's configurable interconnect. A placement solution is *legal* when each netlist primitive occupies a compatible site within a configurable block operating in a compatible mode such that each of its connected nets can reach all terminals within and outside of the block. Ensuring *legality* can amount to a complex intra-block placement and routing problem, which some tools handle in a separate *clustering* step (see Figure 3). Clustering reduces the size and complexity of the placement problem by discretizing the netlist into groups of primitives and nets that are compatible with configurable blocks available on the target device; however, clustering can impede placement quality. Clustering tradeoffs are discussed in Section 2.

The open source Verilog-to-Routing (VTR) [36] tool flow and its place and route tool, VPR, were conceived to support architectural experimentation and innovation [40] [4]. Using VTR's robust architecture modeling language, VPR's flexible and device-agnostic clustering and placement tools can produce a legal solution for any user-defined FPGA architecture with arbitrarily complex configurable block types [31]. VTR is thus widely used for novel architecture development and as an implementation flow targeting commercial, academic, and application-specific devices [35] [43] [12].

VTR's versatility entails limitations, most significantly a reliance on pre-placement clustering to abstract away complexity, maintain scalability, and enable support for complex device architectures. Clustering for abstraction may not be necessary when targeting simpler devices, and scalable placement techniques such as analytical algorithms can obviate the need for clustering to reduce problem size. While VTR must maintain support for arbitrary complexity,

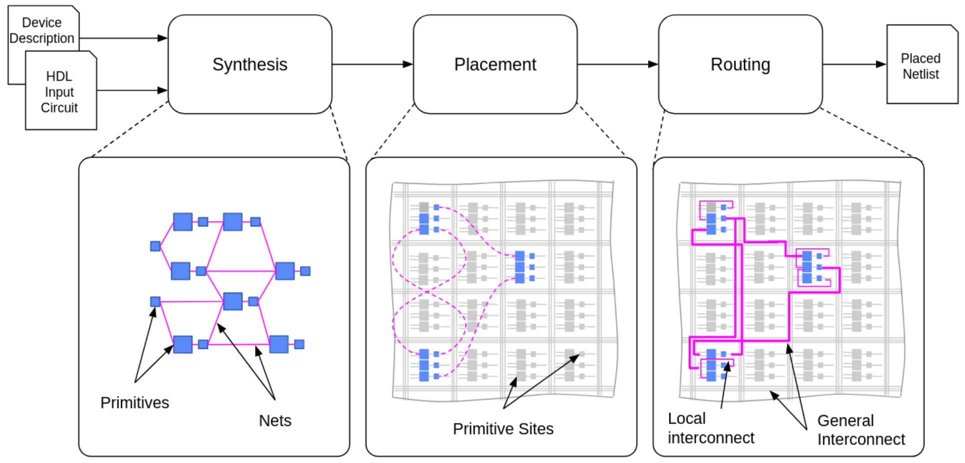


Figure 1: A typical FPGA CAD Flow (image © 2024 Kate Thurmer)

more narrowly scoped academic tools demonstrate that delaying or omitting clustering can improve result quality for some device architectures. By incorporating a variety of external placement tools, VTR can maintain its versatility without sacrificing result quality as FPGAs scale and diversify. Tool developers stand to benefit as well from access to VTR's capabilities (e.g. routing, timing analysis, power estimation). However, while complex, hierarchical device modeling is key to VTR's versatility, it results in a place and route interface requiring a detailed, hierarchical, error-free solution that is beyond the scope of many external tools.

To lower the barrier to mutually beneficial interoperability, we introduce a **V**TR **I**nterface for **P**lacement with **E**rror **R**esilience (*VIPER*). *VIPER* constructs a complete, legal, VTR-compatible clustering and placement solution based on a simplified and potentially illegal externally generated input placement. We demonstrate *VIPER*'s capabilities by (1) accurately reconstructing known legal placements, (2) re-targeting placements from one device architecture to another, and (3) diagnosing legality failure patterns in externally generated placements.

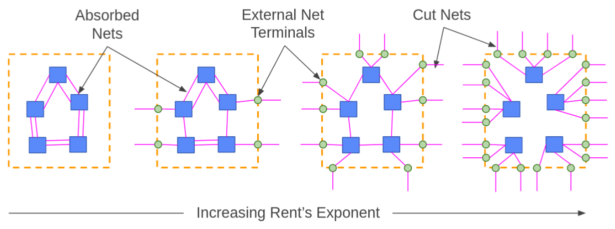


Figure 2: Rent's Exponent for FPGA Clusters (image © 2024 Kate Thurmer)

2 BACKGROUND

A typical FPGA CAD flow consists of *synthesis*, *placement*, and *routing* (see Figure 1). *Clustering* is an optional pre-placement step that reduces problem size and abstracts away low-level complexity by discretizing the netlist into groups of primitives and nets that can be legally placed and routed within an available configurable block type operating in a compatible mode (see Figure 3). Clustering leverages the innate locality of the netlist, grouping together tightly connected primitives so that their shared nets can exploit fast intra-block interconnect and reduce routing demand. However, [13] shows that overly dense clustering of a heavily connected netlist can increase routing demand. To control routing demand, some clustering tools impose limits based on the *Rent's exponent* [25].

A cluster's Rent's exponent is derived from the log-log relationship between the number of external net terminals and the number of primitives in the cluster. As illustrated in Figure 2, for a given cluster size and average net connections per primitive, a lower Rent's exponent corresponds to more nets that are *absorbed* into the cluster and fewer external net terminals. Adding a primitive may reduce or increase a cluster's Rent's exponent, depending on whether its connected nets are absorbed or newly cut. Each cluster's Rent's exponent is upper bounded by the Rent's exponent of the physical configurable block type on which it will be placed [41].

Some aggregative tools (which combine primitives into clusters based on an attraction function) limit cluster Rent's exponents by upper bounding cluster size and/or terminal count [41] [44]. [30] applies variable Rent-based bounds that are higher for more timing-critical clusters. VPR's attraction function penalizes cutting nets with terminals in other clusters, since this creates external terminals which cannot later be eliminated through net absorption [31]; VPR also supports a target external pin utilization parameter [36]. Recursive bipartitioning, e.g. in [19], generates a clustering with the theoretical lowest average Rent's exponent for a given netlist. Observing that partitioning does better at higher tree levels and aggregation is better at lower levels, [34] combines them to achieve a lower Rent's exponent than either alone. Similarly, [47] follows coarse bipartitioning with parallel aggregative clustering.

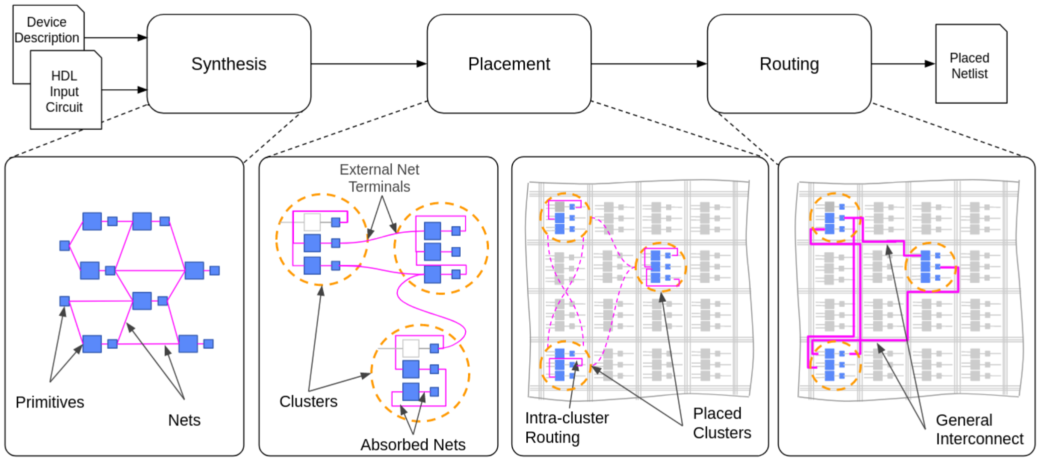


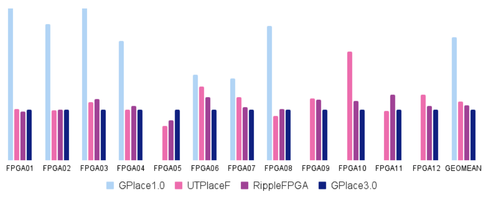
Figure 3: An FPGA CAD Flow with Clustering (image © 2024 Kate Thurmer)

While clustering can maximize net absorption and minimize external net terminals based exclusively on netlist connectivity, accurately prioritizing which nets to absorb requires additional information. Some tools use preliminary timing analysis to estimate net criticality; for example, VPR's attraction function incorporates timing criticality and connectivity [31]. Some (e.g. [7], [42]) generate rough preliminary placements before or during clustering and avoid merging primitives or partial clusters placed far apart. [42], which aggregates primitives into clusters by consensus, uses placement information to suggest mergers based on spatial proximity as well as connectivity. Other tools modify clusters later in the flow, as more accurate placement and routing information becomes available. [45] re-clusters congested regions after routing, [9] allows movement of partial clusters and primitives during annealing-based placement, and [8] duplicates critical primitives during placement in order to straighten and shorten critical paths. [16] enables cluster modification later in the VTR flow.

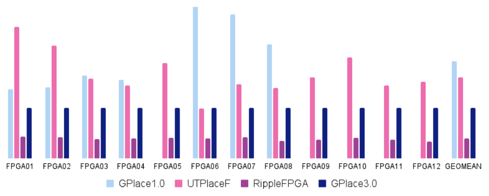
The following comparison illustrates the impact of clustering on placement quality. Analytical Placement (AP) [24] is a scalable, iterative technique in which the placement problem is formulated as a system of equations; rounds of generating a *solved* placement minimizing some objective function, followed by *spreading*, continue until the solved and spread placements converge. AP can operate on clusters, partial clusters, or primitives. The ISPD16 contest framework [50], which is frequently used to evaluate analytical placement tools, includes synthetic benchmark circuits and a simplified device architecture in which configurable blocks are monolithic except *soft logic blocks* (containing LUTs and FFs), which are based on the high Rent's exponent Xilinx Ultrascale CLB [3]; sources of complexity such as carry chains and soft memories are omitted.

AP scalability and the simplicity of the ISPD16 framework render clustering to manage problem size and complexity optional, enabling exploration of the trade space among flexibility, result quality, and runtime. GPlace1.0 [38], a top contest finisher, uses a rough initial placement to inform clustering, then runs AP on a clustered netlist. Newer versions of other top finishers interleave clustering with rounds of AP or delay it until the end of placement.

UTPlaceF [27] begins AP on a *flat* (unclustered) netlist, estimating routing congestion after each round. Near convergence, clusters are aggregated using distance, connectivity, and congestion information; AP then resumes on the clustered netlist, followed by *intra-cluster placement* (assigning primitives to sites within their clusters). RippleFPGA [10] also begins with rounds of AP on a flat netlist, followed by aggregation of *BLEs* (groups of two FFs and a fracturable LUT) based on distance and connectivity. AP resumes on the partially clustered netlist, adding congestion mitigation after each round. After AP, a refinement step allows movement of both clusters and BLEs, followed by intra-cluster placement. GPlace3.0 [1] runs all AP rounds on a flat netlist, adding congestion mitigation in later rounds; post-AP refinement allows movement of both clusters and primitives and minimizes external net terminals. Both RippleFPGA and GPlace3.0 verify cluster legality as BLEs or primitives are moved during spreading and refinement.



(a) Normalized Wirelength (Lower is better)



(b) Normalized Runtime (Lower is better)

Figure 4: ISPD16 Results Normalized to GPlace3.0

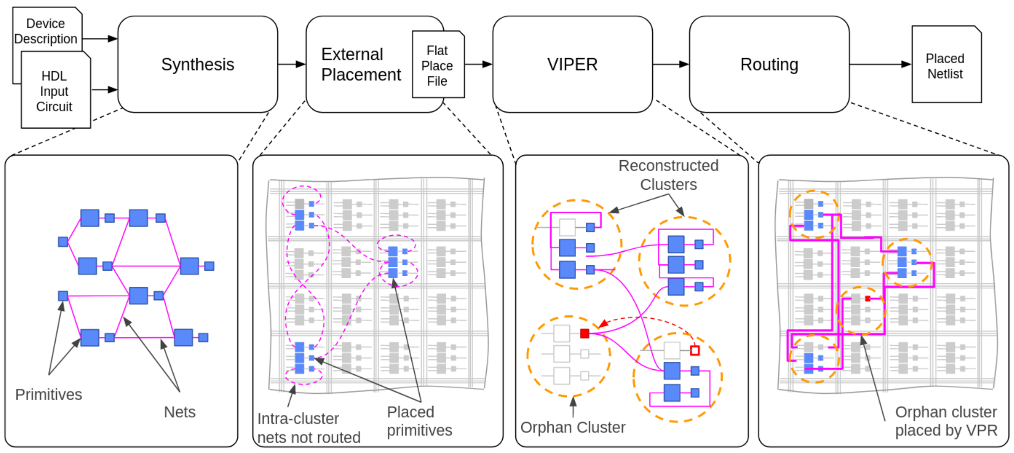


Figure 5: CAD flow with *VIPER* (image © 2024 Kate Thurmer)

Figure 4 shows wirelength and runtime results for the tools described above targeting the ISPD16 framework. Wirelength results generally improve as clustering decisions are finalized later. GPlace3.0, which does not finalize clusters until the end of placement, achieves the lowest wirelength on most benchmarks. RippleFPGA, which interleaves incremental clustering with AP, outperforms UTPlaceF, which finalizes clusters partway through AP. All three outperform GPlace1.0, which finalizes clusters prior to AP. A contributing factor may be that net absorption (a typical clustering objective) is obviated by the target logic block, which offers little intra-cluster connectivity [3]. There is also an apparent tradeoff between flexibility and runtime. Both RippleFPGA and GPlace3.0 move partial clusters or primitives during AP and refinement; this requires repeated legality checking and congestion mitigation – both tasks that might otherwise be handled by pre-placement clustering. RippleFPGA, which operates on BLEs rather than primitives and is thus able to use fast, coarse-grained legality checking, achieves wirelength results nearly comparable to those of GPlace3.0 in half the runtime.

Depending on device size and complexity, clustering can be essential, unnecessary, or even detrimental. Complexity also impacts the suitability of placement techniques. To ensure broad device support, VTR uses a flexible, AI-guided simulated annealing (SA) based placer [17]. Some academic [39] and commercial [22] tool flows maintain scalability and result quality by combining AP with SA refinement. Various AP tools accommodate block diversity in ways that may or may not extend to arbitrary architectures. [20] and [46] interleave rounds of AP on all block types with separate rounds per type; [46] spreads by type and then by column, refining columns using SA. [28] models each block type as a separate system and jointly solves all systems. [23] uses a flexible framework that can model arbitrary device and netlist constraints.

Rather than seeking the ultimate device architecture or the best combination of placement techniques, *VIPER* gives VTR the freedom not to choose. By lowering the barrier to interoperability, *VIPER* encourages tools to join the VTR framework, enabling experimentation with different combinations of algorithms and architectures.

3 *VIPER* OVERVIEW

VIPER is an optional VPR stage run in place of clustering and in place of (or prior to) placement. *VIPER* transforms a simplified, externally generated placement into a legal, VTR-compatible placement solution. Figure 5 shows *VIPER* within an FPGA CAD flow.

3.1 *VIPER* Interface

The current VTR placement and routing interface includes *clustered netlist* (.net) and *clustered placement* (.place) files [15]. Together these files describe a placement solution in terms of VTR's detailed internal device model, including block hierarchy, mode, and intra-cluster connectivity information. This description must be complete and error free to be accepted by VTR, and generating it is beyond the scope of many external placement tools.

Figure 6 illustrates *VIPER*'s simplified interface, which uses a new *flat placement file* format. For each primitive (e.g. LUT, FF, multiplier) in the netlist, the flat placement file lists: the primitive's **name**, **cluster coordinates** for the cluster in which the primitive is placed, and a **primitive site index** indicating the specific location of the primitive within its cluster.

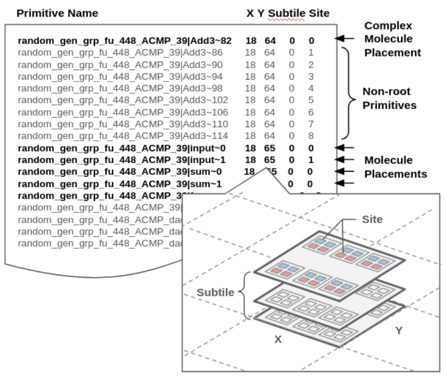
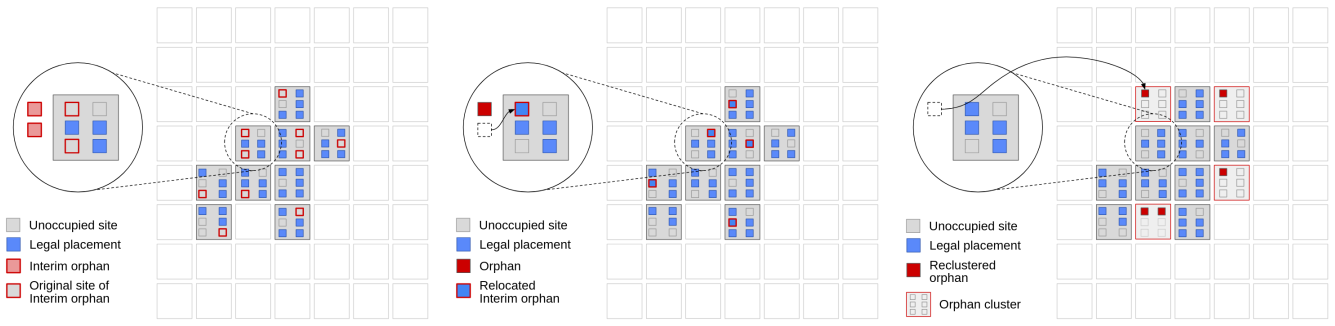

```
Primitive Name                                X  Y  Subtile Site
random_gen_grp_fu_448_ACMP_39|Add3~82         18  64  0  0
random_gen_grp_fu_448_ACMP_39|Add3~86         18  64  0  1
random_gen_grp_fu_448_ACMP_39|Add3~90         18  64  0  2
random_gen_grp_fu_448_ACMP_39|Add3~94         18  64  0  3
random_gen_grp_fu_448_ACMP_39|Add3~98         18  64  0  4
random_gen_grp_fu_448_ACMP_39|Add3~102        18  64  0  5
random_gen_grp_fu_448_ACMP_39|Add3~106        18  64  0  6
random_gen_grp_fu_448_ACMP_39|Add3~110        18  64  0  7
random_gen_grp_fu_448_ACMP_39|Add3~114        18  64  0  8
random_gen_grp_fu_448_ACMP_39|input~0         18  65  0  0
random_gen_grp_fu_448_ACMP_39|input~1         18  65  0  1
random_gen_grp_fu_448_ACMP_39|sum~0           18  65  0  0
random_gen_grp_fu_448_ACMP_39|sum~1                   0  0
random_gen_grp_fu_448_ACMP_3
random_gen_grp_fu_448_ACMP_39
random_gen_grp_fu_448_ACMP_da
random_gen_grp_fu_448_ACMP_da
random_gen_grp_fu_448_ACMP_da
```

Figure 6: *VIPER* Flat Placement File Format (image © 2024 Kate Thurmer)



(a) Cluster Reconstruction **(b) Cluster Repair** **(c) Orphan Reclustering**

Figure 7: *VIPER* Cluster Reconstruction and Repair (images © 2024 Kate Thurmer)

3.1.1 Inputs. *VIPER* inputs are a synthesized **netlist file**, a **flat placement file**, and a VTR-compatible **XML architecture file** [14] specifying a fixed target device layout; flat placement file site indices must correspond to complex block descriptions in this file.

3.1.2 Outputs. *VIPER* outputs a **flat placement file** and VTR-compatible **.net** and **.place files**. These files can be used as inputs to VPR routing or as a starting point for refinement using VPR's simulated annealing (SA) based placer.

3.1.3 API Functions. *VIPER*'s cluster reconstruction and legality checking functions form an extension to the VPR re-clustering API [16]. These functions are described in Section 3.3.3.

3.2 *VIPER* Flow

3.2.1 Pre-clustering. *VIPER* first pre-processes the input netlist, invoking VPR pre-clustering [32] to form *complex molecules*, which are groups of connected netlist primitives (e.g. carry chains) with fixed locations relative to a designated *root primitive*. Per VPR convention, each primitive that is not part of a complex molecule is considered the root of a molecule of size one. We refer to the placement coordinates of a molecule's root primitive as the *molecule placement*; all primitives in a molecule are placed as a unit according to the molecule placement, and *VIPER* ignores flat placement file entries for non-root primitives.

3.2.2 Cluster Reconstruction. Figure 7a illustrates a simplified example of cluster reconstruction. *VIPER* reconstructs and places clusters based on an input flat placement, inferring mode information and optionally using VPR's pin counting based legality checking [32] as each molecule is added to its target cluster (if the input placement is presumed to be legal, the user may opt to skip pin counting). *VIPER* attempts to place each molecule on the precise primitive site(s) specified in the input placement; upon pin counting failure or mode conflict, *VIPER* designates the molecule an *interim orphan* (our term for a molecule that failed to be placed on its designated primitive site in its original cluster) and skips it.

3.2.3 Cluster Legality Checking. After reconstruction, *VIPER* subjects each reconstructed input cluster to an intra-cluster routing-based legality check [32].

3.2.4 Cluster Repair. Figure 7b illustrates a simplified example of cluster repair. If a cluster that *VIPER* has reconstructed per Section 3.2.2 fails the legality check in Section 3.2.3, *VIPER* dissolves the cluster and attempts to construct a (complete or partial) legal version of it. *VIPER* first attempts to place each molecule on its original primitive site(s), checking cluster legality using intra-cluster routing (preceded by pin counting if it was skipped earlier) after each molecule. Each molecule that fails a legality check is designated an *interim orphan* and skipped. After attempting to place each molecule on its original site, *VIPER* attempts to place interim orphans on any open sites in the cluster. Interim orphans that are not successfully placed during cluster repair are designated *orphans* (our term for a molecule that cannot join its original cluster).

3.2.5 Orphan Reclustering. Figure 7c illustrates a simplified example of orphan reclustering. If orphans remain after cluster repair, *VIPER* forms them into one or more new *orphan clusters*. *VIPER* can be configured to (1) form an individual new cluster for each orphan, (2) form an annex cluster for the orphans from each original cluster, or (3) greedily form dense clusters from all orphans – subject to an optional maximum cluster size; these options are illustrated in Figure 8. While creating individual or annex orphan clusters may enable orphans to be placed closer to their original clusters, dense orphan clustering may be useful when there are many orphans or space on the target device is limited.

3.2.6 Orphan Cluster Placement. *VIPER* assigns placement coordinates to all reconstructed input clusters according to the input placement, then uses VPR's simulated annealing (SA) based placer to place orphan clusters. After orphan reclustering, VPR assigns initial placement coordinates to all orphan clusters and then iteratively refines their placement using SA (while holding all reconstructed input clusters fixed). Alternatively, VPR can refine the entire input placement by allowing movement of both orphan clusters and reconstructed input clusters during SA placement.

3.2.7 Diagnostics. During reconstruction and repair, *VIPER* logs legality check failures along with diagnostic information, such as intra-cluster routing congestion, routing failures, and mode conflicts, that can help external tool developers to identify failure patterns. We show an example in Section 4.4.

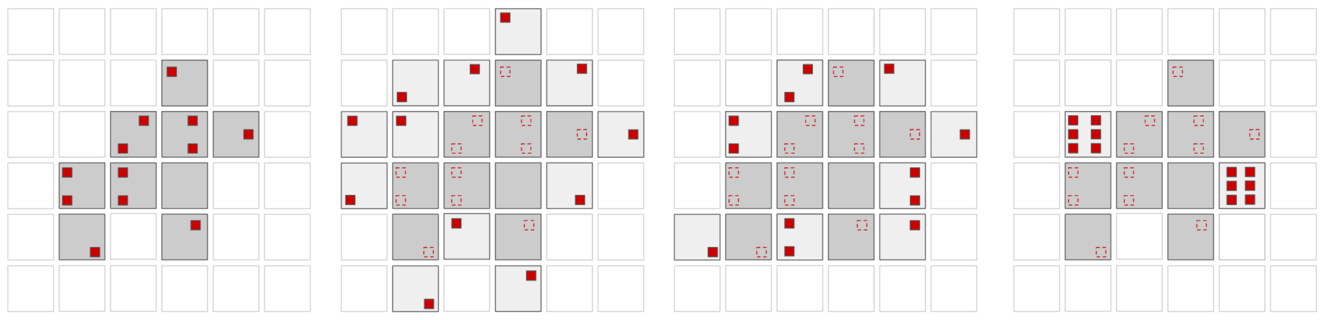

(a) Original Orphan Placement **(b) Individual Orphan Clusters** **(c) Annex Orphan Clusters** **(d) Dense Orphan Clusters**

Figure 8: Orphan Reclustering and Placement (images © 2024 Kate Thurmer)

3.3 Augmented VPR Re-Clustering API

VPR's re-clustering API [16] enables modification of an existing, complete clustered netlist while maintaining VPR's legality guarantee. *VIPER*'s related use case is to construct and verify the legality of a clustered netlist from molecules not previously clustered, based on externally generated intra-cluster molecule placements. We contribute the following enhancements to the API:

3.3.1 Force Site Option. When adding a molecule to a new or existing cluster, a new optional *force site* parameter instructs VPR to attempt to place the molecule root on a specified primitive site in the cluster and to report failure if the placement is unsuccessful. If no forced site is specified, VPR selects a legal site (if available).

3.3.2 Legality Checking Options. The re-clustering API uses intra-cluster routing to check legality each time a molecule is added to or removed from a cluster, ensuring that the move does not destroy the legality guarantee of the clustering. *VIPER* builds clusters one molecule at a time, and so its runtime is significantly reduced by using faster pin counting based incremental legality checks [32] after each addition, checking intra-cluster routing only after each cluster is complete. When *VIPER* reconstructs a presumed legal input placement (see Section 4.1), even fast incremental legality checks may be unnecessary. New optional parameters enable or disable pin counting and/or routing-based legality checking when adding a molecule to a cluster.

3.3.3 VIPER API Functions. The re-clustering API relies on the assumption that a molecule is always moved from an existing cluster (for which the type and mode are known) into a new or existing cluster. To increase flexibility, we extend the API to include *VIPER* functions to:

- add a not-yet-clustered molecule to an existing or newly created cluster, with force site and incremental legality checking options as described in Sections 3.3.1 and 3.3.2; when creating a new cluster, *VIPER* infers cluster type and mode and optionally assigns user-specified placement coordinates;
- request an intra-cluster routing-based legality check on a proposed cluster as described in Section 3.2.3;
- dissolve and attempt to reconstruct a proposed cluster as described in Section 3.2.4.

4 DEMONSTRATING *VIPER*

Below we demonstrate several *VIPER* use cases. All experiments were run on a virtual machine with an AMD EPYC 7543, 60 GB RAM equivalent architecture.

4.1 Reconstructing Legal Placements

We show that *VIPER* can reconstruct known legal placements without information loss or quality degradation.

4.1.1 Reconstructing RippleFPGA ISPD16 Placements. We use RippleFPGA [51] to generate placements for the ISPD16 contest framework [50], convert them to *VIPER*'s flat placement file format, reconstruct with *VIPER* (using open source, VTR-compatible versions of the ISPD16 contest device architecture [49] and benchmark circuits [48]), convert back to ISPD16 format, and route using Vivado with an ISPD16 contest patch [50]. The *VIPER*-reconstructed placement files are identical to their corresponding RippleFPGA solution files for all circuits. Table 1 shows runtime (RT) and Vivado routed wirelength (WL) for our RippleFPGA solutions, *VIPER* reconstructions, and published RippleFPGA results [51]. Our pre- and post-*VIPER* Vivado WL results are identical to one another and similar to the published results (the ISPD16 contest does not include a timing metric). *VIPER* runtimes shown are for reconstructing all block types, including monolithic RAM and DSP blocks.

4.1.2 Reconstructing VPR Titan23/StratixIV Placements. We use VPR to generate placements for the Titan23 benchmark circuits [37] targeting an open source StratixIV-like device architecture [49], export the VPR placements in both VTR (.net and .place) and *VIPER* (flat placement file) formats, reconstruct them from flat placement files with *VIPER*, and export reconstructed placements in both formats. The original and *VIPER*-reconstructed flat placement files are identical for all circuits. We generate estimated routing results with VPR placement and use VPR routing to route pre- and post-*VIPER* solutions. Table 2 compares VPR post-placement estimated and actual post-routing critical path delay (CPD) and routed wirelength (WL) for original and *VIPER*-reconstructed solutions, along with runtime (RT) for VPR clustering and placement, *VIPER* reconstructing soft logic only, and *VIPER* reconstructing all blocks – including complex RAM and DSP blocks.

Table 1: Reconstructing RippleFPGA Placements with *VIPER*

ISPD16 Circuit	**RippleFPGA Placement**		***VIPER* Reconstruction**		**RippleFPGA Published Results** [51]	
	Placement RT (m:s)	Vivado WL x10^6	*VIPER* RT (m:s)	Vivado WL x10^6	Placement RT (m:s)	Vivado WL x10^6
FPGA01	0:36	0.353	0:28	0.353	0:38	0.35
FPGA02	1:04	0.646	0:46	0.646	0:59	0.64
FPGA03	4:00	3.337	1:53	3.337	3:52	3.25
FPGA04	4:38	5.387	1:51	5.387	5:21	5.49
FPGA05	5:42	9.909	1:52	9.909	5:57	9.91
FPGA06	9:08	6.277	3:12	6.277	9:08	6.14
FPGA07	9:33	9.791	3:07	9.791	10:54	9.59
FPGA08	7:23	9.080	3:22	9.080	7:01	8.09
FPGA09	10:59	12.259	3:58	12.259	11:21	12.06
FPGA10	20:06	7.077	4:10	7.077	17:26	6.97
FPGA11	11:30	11.017	3:50	11.017	10:55	10.92
FPGA12	13:02	7.122	5:03	7.122	13:18	7.24

Table 2: Reconstructing VPR Placements with *VIPER*

Titan23 Circuit (*% soft logic*)	**VPR Clustering & Placement**					***VIPER* Reconstruction**					
		VPR Placer est.		VPR Router		*Soft Logic*	*All Blocks*	VPR Placer est.		VPR Router	
	RT (m:s)	WL x10^6	CPD (s)	WL x10^6	CPD (s)	RT (m:s)	RT (m:s)	WL x10^6	CPD (s)	WL x10^6	CPD (s)
neuron (95)	8:36	0.61	7.36	0.78	7.63	1:02	5:42	0.61	7.36	0.78	7.68
sparcT1 core (95)	8:05	0.93	7.52	1.27	8.13	1:22	8:19	0.93	7.52	1.27	8.14
stereovision (95)	8:40	0.50	7.20	0.58	7.41	1:00	5:56	0.50	7.20	0.58	7.41
cholesky mc (95)	12:27	0.80	6.37	1.14	6.98	1:21	8:47	0.80	6.37	1.14	7.12
des90 (85)	27:33	1.63	11.65	2.24	12.49	1:30	31:02	1.63	11.65	2.24	12.58
SLAM spheric (97)	14:38	1.23	78.26	1.61	77.48	1:46	2:03	1.23	78.40	1.61	77.41
segmentation (97)	26:08	1.31	856.47	1.71	851.68	1:58	14:25	1.31	856.47	1.71	851.93
bitonic mesh (83)	55:16	3.57	13.23	4.66	14.03	2:33	59:15	3.57	13.12	4.67	14.04
dart (94)	23:38	1.95	13.08	2.29	14.07	2:53	21:46	1.95	13.08	2.28	14.07
openCV (92)	40:31	2.60	9.22	3.28	9.84	2:43	40:56	2.60	9.39	3.27	10.07
stap qrd (99)	40:51	2.20	7.63	2.65	8.40	3:06	12:51	2.20	7.63	2.65	8.40
minres (93)	43:47	2.10	7.52	2.76	10.22	3:10	37:04	2.11	7.69	2.76	11.91
cholesky bdti (98)	30:15	2.03	8.14	2.67	8.55	3:05	5:44	2.03	8.14	2.67	8.55
sparcT2 core (97)	45:59	3.76	10.08	4.76	10.87	4:24	16:36	3.76	9.98	4.76	10.62
denoise (96)	53:31	2.40	853.16	3.02	846.10	3:56	21:41	2.40	853.16	3.02	845.79
gsm switch (93)	83:52	4.58	8.91	5.43	9.36	5:58	61:12	4.58	9.08	5.42	9.56
mes noc (95)	96:21	4.08	10.63	5.07	11.69	8:19	40:19	4.08	10.63	5.07	11.42
LU230 (88)	150:41	16.91	22.55	18.02	22.60	5:46	131:15	16.92	22.55	18.02	22.63
LU Network (94)	85:51	4.81	8.87	5.86	9.85	7:10	73:04	4.82	8.87	5.87	9.85
sparcT1 chip2 (98)	131:29	6.43	16.60	7.58	17.33	10:27	34:14	6.43	16.60	7.58	17.20
directrf (98)	177:22	11.65	9.66	12.47	10.47	10:43	72:03	11.65	9.66	12.46	10.32
bitcoin miner (94)	148:47	9.33	7.22	10.83	9.35	10:06	73:46	9.33	7.36	10.80	9.22
gaussianblur (> 99)	651:04	40.55	828.96	32.41	820.95	30:49	30:44	40.55	828.96	32.17	821.14

Table 3: Retargeting RippleFPGA Placements with *VIPER*

ISPD Circuit	RippleFPGA % Clusters with Legality Violations	*VIPER* Retargeting & Repair: % Clusters Fully Repaired, Num. Violations in Cluster 1	2	3+	Total	% Soft Logic Molecules Orphaned	*VIPER* RT (m:s) *(without orphan reclustering)*
FPGA01	13.05	88.17	64.29	40.91	83.91	0.193	1:23
FPGA02	10.68	93.09	85.51	50.00	92.17	0.068	2:03
FPGA03	25.35	96.21	91.62	60.71	95.50	0.099	5:35
FPGA04	26.19	95.06	89.24	68.33	94.17	0.138	5:31
FPGA05	25.26	95.62	87.30	56.10	94.53	0.145	5:35
FPGA06	28.75	92.34	76.82	59.12	89.29	0.273	9:27
FPGA07	28.71	91.76	75.73	53.63	88.73	0.300	9:27
FPGA08	12.67	94.81	87.93	50.00	94.36	0.065	8:29
FPGA09	25.09	89.55	79.18	65.07	87.92	0.238	11:27
FPGA10	42.48	77.42	51.96	23.43	67.95	1.263	13:31
FPGA11	18.70	87.58	72.34	40.94	85.17	0.227	10:47
FPGA12	37.12	73.03	44.37	18.61	63.61	1.006	15:30
Geo.Mean:	22.63	89.26	73.89	45.91	85.80	0.215	6:47

4.2 Retargeting and Repair

We demonstrate *VIPER*'s ability to reconstruct and repair a not entirely legal placement by *retargeting*, i.e. transforming a placement intended for one device architecture such that it is legal on another. We retarget RippleFPGA [10] placements intended for the ISPD16 Ultrascale-like device architecture to an open source StratixIV-like device architecture [49] with the following modifications: (1) eight BLEs per soft logic block – the Ultrascale CLB has eight BLEs, and so retargeting to a typical StratixIV logic block with ten is too easy; (2) monolithic RAM and DSP blocks as in the ISPD16 framework; and (3) a fixed layout matching that of the ISPD16 device architecture. Table 3 shows detailed results. On average over all circuits, 23% of input clusters have *legality violations* (illegal molecule placements); *VIPER fully repairs* (relocates all interim orphans to legal sites within the cluster) over 85% of them, repairing 89%, 74% and 46% of clusters with 1, 2, and 3+ violations, respectively.

4.3 Orphan Reclustering and Placement

In the experiment in Section 4.2, on average 0.2% of soft logic molecules are orphaned. We conduct three experiments, using *VIPER* to re-cluster these orphans into: (1) individual clusters, (2) annex clusters, and (3) dense clusters. In each experiment, *VIPER* places both fully and *partially repaired* (missing some orphaned molecules) input clusters per the input placement, then VPR places the orphan clusters and refines their placement with SA. We then convert the complete placement solutions to ISPD format and route using Vivado with an ISPD16 contest patch [50]. Table 4 shows percent increase in cluster count and Vivado routed wirelength (WL) compared to the original RippleFPGA results in Table 1, along with runtime for *VIPER* and for VPR placement. Two of the largest circuits only fit on the target device when we use dense orphan reclustering, which has the least impact on cluster count, whereas the sparser individual and annex reclustering options create more cluster but have less impact on wirelength.

4.4 Diagnosing Failure Patterns

In Section 4.2, RippleFPGA placements are retargeted from an Ultrascale-like [3] to a modified StratixIV-like [21] device architecture, the soft logic blocks of which have subtle connectivity differences. In each, a BLE contains one fracturable LUT and two FFs. StratixIV limits total BLE data inputs (shared among LUT and FFs) to eight, while Ultrascale only limits LUT inputs. For each legality violation, *VIPER* reports an intra-cluster routing failure due to congestion at a BLE data input pin; analysis confirms that failure occurs when a BLE has more than eight data inputs. A joint constraint on LUT and FF inputs would enable RippleFPGA to target both architectures; however, since *VIPER* can repair on average 85% of clusters with such violations at a wirelength cost under 4%, a new constraint may not be worth the effort or tool runtime cost. An alternative is to modify an existing constraint (e.g. LUT inputs or FF density) and rely on *VIPER* to repair remaining violations.

5 CONCLUSION

We introduce and demonstrate *VIPER*, a new VTR interface that constructs a complete, legal, and VTR-compatible clustering and placement solution based on a simplified and potentially illegal input placement. By lowering the barrier to interoperability, *VIPER* will enable more tools to join the VTR framework, fostering experimentation with different combinations of placement algorithms and FPGA architectures. Future work includes (1) exploring more robust methods for legality checking, cluster repair, and orphan reclustering and (2) developing a mapping among architecture and circuit characteristics and clustering and placement methods.

ACKNOWLEDGMENTS

The authors thank Mohamed Elgammal for support in using the re-clustering API; Alexandre Singer, Matthew Harger, and Jason Anderson for helpful discussions; the Intel/VMWare Crossroads 3D-FPGA Academic Research Centre and NSERC for funding support.

Table 4: Orphan Re-Clustering for Retargeted RippleFPGA Placements with *VIPER*

ISPD Circuit	*VIPER*: **Individual Orphan Clusters**				*VIPER*: **Annex Orphan Clusters**				*VIPER*: **Dense Orphan Clusters** *Max Size = 8*			
	% New Clusters	% WL Increase	*VIPER* RT (m:s)	VPR Placer RT(m:s)	% New Clusters	% WL Increase	*VIPER* RT (m:s)	VPR Placer RT(m:s)	% New Clusters	% WL Increase	*VIPER* RT (m:s)	VPR Placer RT(m:s)
FPGA01	2.39	1.24	1:24	2:27	2.10	1.09	1:23	2:35	0.31	3.13	1:33	4:15
FPGA02	0.85	0.67	2:00	4:36	0.84	0.90	2:03	4:39	0.11	2.86	2:13	17:03
FPGA03	1.17	2.05	5:32	18:41	1.14	2.18	5:27	19:07	0.15	3.74	6:08	64:03
FPGA04	1.58	2.31	5:32	18:42	1.53	2.17	5:28	19:03	0.20	2.80	6:25	58:39
FPGA05	1.43	1.19	5:32	20:56	1.38	1.08	5:36	21:24	0.18	2.02	6:32	68:05
FPGA06	3.49	16.59	9:14	36:38	3.08	16.44	9:33	38:07	0.44	11.90	12:11	80:26
FPGA07	3.65	11.79	9:16	36:03	3.24	11.09	9:34	38:17	0.46	8.68	12:12	77:30
FPGA08	0.74	2.07	8:22	35:16	0.71	2.11	8:30	34:40	0.09	1.69	9:12	107:04
FPGA09	3.19	14.44	11:14	45:09	3.03	15.09	11:27	43:32	0.40	9.05	14:32	76:11
FPGA10	18.97	N/A*	13:20	N/A*	13.62	N/A*	13:52	N/A*	2.37	56.84**	21:12	62:04
FPGA11	3.03	11.75	10:40	32:39	2.77	11.59	10:49	31:42	0.38	6.79	13:33	44:27
FPGA12	17.37	N/A*	15:25	N/A*	13.51	N/A*	16:06	N/A*	2.17	77.25**	25:18	60:10
Geo.Mean	2.72	3.55	6:43	0.01	2.45	3.57	6:50	0.01	0.34	4.27	8:25	48:10

**VIPER*-generated clustered netlist does not fit on the target device – cannot place or route. **Excluded from geo. mean computation.

REFERENCES

[1] Ziad Abuowaimer, Dani Maarouf, Timothy Martin, Jeremy Foxcroft, Gary Gréwal, Shawki Areibi, and Anthony Vannelli. 2018. GPlace3. 0: Routability-driven analytic placer for UltraScale FPGA architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 5 (2018), 1–33.

[2] Sassan Ahmadi. 2016. *Toward 5G Xilinx Solutions and Enablers for Next-Generation Wireless Systems.* Technical Report WP476. Xilinx Inc. https://docs.xilinx.com/v/u/en-US/wp476-toward-5g

[3] AMD. 2022. Ultrascale Architecture and product data sheet: Overview. Retrieved January 2024 from https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview

[4] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications.* Springer, 213–222.

[5] Andrew Boutros and Vaughn Betz. 2021. FPGA Architecture: Principles and Progression. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 4–29. https://doi.org/10.1109/MCAS.2021.3071607

[6] 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 *2020 International Conference on Field-Programmable Technology (ICFPT).* 10–19. https://doi.org/10.1109/ICFPT51103.2020.00011

[7] Doris T. Chen, Kristofer Vorwerk, and Andrew Kennings. 2007. Improving Timing-Driven FPGA Packing with Physical Information. In *2007 International Conference on Field Programmable Logic and Applications.* 117–123. https://doi.org/10.1109/FPL.2007.4380635

[8] Gang Chen and Jason Cong. 2004. Simultaneous Placement with Clustering and Duplication. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) *(DAC '04).* Association for Computing Machinery, New York, NY, USA, 740–772.

[9] Gang Chen and Jason Cong. 2004. Simultaneous timing driven clustering and placement for FPGAs. In *International Conference on Field Programmable Logic and Applications.* Springer, 158–167.

[10] Gengjie Chen, Chak-Wa Pui, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Evangeline F. Y. Young, and Bei Yu. 2018. RippleFPGA: Routability-Driven Simultaneous Packing and Placement for Modern FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37, 10 (2018), 2022–2035.

[11] D. Chiou. 2017. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC).* IEEE Computer Society, Los Alamitos, CA, USA, 124–124. https://doi.org/10.1109/IISWC.2017.8167769

[12] DARPA. 2016. A Camera That Can See Unlike Any Imager Before It. https://www.darpa.mil/news-events/2016-09-16

[13] André DeHon. 1999. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '99).* Association for Computing Machinery, New York, NY, USA, 69–78. https://doi.org/10.1145/296399.296431

[14] VTR Documentation. 2022. FPGA Architecture Description. https://docs.verilogtorouting.org/en/latest/arch/#fpga-architecture-description

[15] VTR Documentation. 2022. VPR File Formats. https://docs.verilogtorouting.org/en/latest/vpr/file_formats

[16] Mohamed A. Elgammal and Vaughn Betz. 2023. Breaking Boundaries: Optimizing FPGA CAD with Flexible and Multi-Threaded Re-Clustering. In *Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies* (Kusatsu, Japan) *(HEART '23).* Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/3597031.3597054

[17] 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 (2022), 2532–2545.

[18] Rory Fagan, Frank C. Robey, and Luke Miller. 2018. Phased array radar cost reduction through the use of commercial RF systems on a chip. In *2018 IEEE Radar Conference (RadarConf18).* 0935–0939. https://doi.org/10.1109/RADAR.2018.8378686

[19] Wenyi Feng, Jonathan Greene, Kristofer Vorwerk, Val Pevzner, and Arun Kundu. 2014. Rent's Rule Based FPGA Packing for Routability Optimization. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14).* Association for Computing Machinery, New York, NY, USA, 31–34. https://doi.org/10.1145/2554688.2554763

[20] Marcel Gort and Jason H. Anderson. 2012. Analytical placement for heterogeneous FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL).* 143–150. https://doi.org/10.1109/FPL.2012.6339278

[21] Intel. 2016. Stratix-IV Features. https://www.intel.com/content/www/us/en/products/programmable/devices/features.html

[22] Intel. 2022. Intel Quartus Prime Pro Edition User Guide Version 22.1 (ID 683236). https://www.intel.com/content/www/us/en/docs/programmable/683236/22-1/design-place-and-route.html

[23] Nima Karimpour Darav, Andrew Kennings, Kristofer Vorwerk, and Arun Kundu. 2019. Multi-Commodity Flow-Based Spreading in a Commercial Analytic Placer. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19).* Association for Computing Machinery, New York, NY, USA, 122–131. https://doi.org/10.1145/3289602.3293896

[24] Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov. 2012. SimPL: An Effective Placement Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 1 (2012), 50–60. https://doi.org/10.1109/TCAD.2011.2170567

[25] B.S. Landman and R.L. Russo. 1971. On a Pin Versus Block Relationship For Partitions of Logic Graphs. *IEEE Trans. Comput.* C-20, 12 (1971), 1469–1479. https://doi.org/10.1109/T-C.1971.223159

[26] Vasileios Leon, George Lentaris, Dimitrios Soudris, Simon Vellas, and Mathieu Bernou. 2022. Towards Employing FPGA and ASIP Acceleration to Enable Onboard AI/ML in Space Applications. In *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. 1–4. https://doi.org/10.1109/VLSI-SoC54400.2022.9939566

[27] Wuxi Li, Shounak Dhar, and David Z Pan. 2017. UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37, 4 (2017), 869–882.

[28] Wuxi Li, Yibo Lin, and David Z Pan. 2019. elfPlace: Electrostatics-based Placement for Large-Scale Heterogeneous FPGAs. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2019), 1–8.

[29] Chao Liu, Michael E Jones, and Angela C Taylor. 2020. Characterizing the performance of high-speed data converters for RFSoC-based radio astronomy receivers. *Monthly Notices of the Royal Astronomical Society* 501, 4 (12 2020), 5096–5104. https://doi.org/10.1093/mnras/staa3895

[30] Hanyu Liu and Ali Akoglu. 2010. Timing-driven nonuniform depopulation-based clustering. *International Journal of Reconfigurable Computing* 2010 (2010). https://doi.org/10.1155/2010/158602

[31] Jason Luu, Jason Helge Anderson, and Jonathan Scott Rose. 2011. Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 227–236. https://doi.org/10.1145/1950413.1950457

[32] Jason Luu, Jonathan Rose, and Jason Anderson. 2014. Towards Interconnect-Adaptive Packing for FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 21–30.

[33] R. Maiden, C. Lanzani, and A. Vora. 2023. *Build More Cost-Effective and More Efficient 5G Radios with Intel Agilex FPGAs*. Technical Report WP-01312-1.1. Intel Corporation. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/build-5g-radios-with-agilex-fpgas-white-paper.pdf

[34] Zied Marrakchi, Hayder Mrabet, and Habib Mehrez. 2006. Evaluation of Hierarchical FPGA partitioning methodologies based on architecture Rent Parameter. In *2006 Ph. D. Research in Microelectronics and Electronics*. IEEE, 85–88.

[35] 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 (July 2020), 49–57.

[36] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, and *et al.* 2020. VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling. *ACM Trans. Reconfigurable Technol. Syst.* 13, 2, Article 9 (May 2020), 55 pages.

[37] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2013. Titan: Enabling large and complex benchmarks in academic CAD. In *23rd International Conference on Field programmable Logic and Applications*. 1–8. https://doi.org/10.1109/FPL.2013.6645503

[38] Ryan Pattison, Ziad Abuowaimer, Shawki Areibi, Gary Gréwal, and Anthony Vannelli. 2016. GPlace: A congestion-aware placement tool for UltraScale FPGAs. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7. https://doi.org/10.1145/2966986.2980085

[39] Rachel Selina Rajarathnam, Kate Thurmer, David Pan, Vaughn Betz, and Mahesh A. Iyer. 2024. Better Together: Combining Analytical and Annealing Methods for FPGA Placement [Manuscript under review]. In *to appear in 34rd International Conference on Field programmable Logic and Applications*.

[40] 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 *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. 77–86.

[41] Amit Singh and Malgorzata Marek-Sadowska. 2002. Efficient Circuit Clustering for Area and Power Reduction in FPGAs. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '02)*. Association for Computing Machinery, New York, NY, USA, 59–66. https://doi.org/10.1145/503048.503058

[42] Love Singhal, Mahesh A. Iyer, and Saurabh Adya. 2017. LSC: A Large-Scale Consensus-Based Clustering Algorithm for High-Performance FPGAs. *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Article 30 (2017), 6 pages.

[43] Xifan Tang, Edouard Giacomin, Baudouin Chauviere, Aurelien Alacchi, and Pierre-Emmanuel Gaillardon. 2020. OpenFPGA: An open-source framework for agile prototyping customizable FPGAs. *IEEE Micro* 40, 4 (2020), 41–48.

[44] Russell Tessier and Heather Giza. 2000. *Balancing Logic Utilization and Area Efficiency in FPGAs*. Technical Report TR-CSE-00-5. Department of Electrical and Computer Engineering, University of Massachusetts.

[45] Marvin Tom, David Leong, and Guy Lemieux. 2006. Un/DoPack: Re-Clustering of Large System-on-Chip Designs with Interconnect Variation for Low-Cost FPGAs. In *2006 IEEE/ACM International Conference on Computer Aided Design*. 680–687. https://doi.org/10.1109/ICCAD.2006.320013

[46] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. 2017. Liquid: High quality scalable placement for large heterogeneous FPGAs. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 17–24. https://doi.org/10.1109/FPT.2017.8280116

[47] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. 2018. How Preserving Circuit Design Hierarchy During FPGA Packing Leads to Better Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 3 (2018), 629–642. https://doi.org/10.1109/TCAD.2017.2717786

[48] VTR. 2023. ISPD16 Blif Files. https://github.com/verilog-to-routing/vtr-verilog-to-routing/blob/master/vtr_flow/benchmarks/ispd-blif/

[49] VTR. 2023. Open Source FPGA Architecture Models. https://github.com/verilog-to-routing/vtr-verilog-to-routing/blob/master/vtr_flow/arch/

[50] Stephen Yang, Aman Gayasen, Chandra Mulpuri, Sainath Reddy, and Rajat Aggarwal. 2016. Routability-driven FPGA placement contest. *Proc. of the International Symposium on Physical Design (ISPD)* (2016), 139–143.

[51] Evangaline Young. 2019. RippleFPGA. https://github.com/cuhk-eda/ripple-fpga.