HETRIS: Adaptive Floorplanning for Heterogeneous FPGAs

Kevin E. Murray and Vaughn Betz
Department of Electrical and Computer Engineering
University of Toronto, Ontario, Canada
Email: {kmurray,vaughn}@eecg.utoronto.ca

***Abstract*—Floorplanning is an approach to improve the scalability of existing CAD algorithms, facilitate team-based design, and also plays an important role in partial reconfiguration. This work introduces HETRIS, a new automated floorplanning tool for heterogeneous FPGAs. HETRIS uses an adaptive legality approach to target arbitrary FPGA architectures. It includes enhancements enabling it to run on average $15.6\times$ faster than previous work, while producing denser floorplans than a commercial tool. Using HETRIS we perform the first evaluation of an FPGA floorplanner using real-world benchmarks, allowing us to investigate the relationship between partitioning, floorplanning and FPGA architecture.**

I. INTRODUCTION

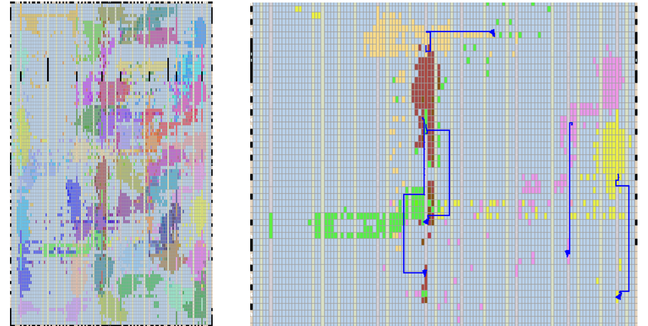
Designing FPGA-based systems is a time-consuming process, particularly as ever increasing design sizes lead to multi-hour CAD run-times [1], and many design iterations. Floorplanning, which allocates specific parts of a design to dedicated regions on a device, is one approach to address these challenges. Floorplanning enables a divide-and-conquer approach to the physical implementation of large designs by decoupling them spatially. This enables efficient parallel compilation, and is also beneficial for team-based design, enabling independent design and optimization before final integration. Floorplanning is also an important part of partial reconfiguration design flows. In this paper we:

- Present HETRIS[1], a new automatic floorplanning tool targeting arbitrary FPGA architectures with an adaptive and scalable legality approach,
- Investigate the structure of the FPGA floorplanning solution space and its relation to FPGA architecture,
- Perform the first evaluation of FPGA floorplanning using realistic benchmarks, and
- Compare HETRIS to a commercial tool when floorplanning at high resource utilization.

II. FLAT COMPILATION VS. FLOORPLANNING

In the conventional FPGA CAD flow, compilation is performed in a 'flat' manner, where both logical and physical synthesis can optimize across the entire design[2]. Flat compilation gives tools global visibility, potentially enabling better optimization results. However, given the large solution space and heuristic nature of CAD tools this may not produce the best result. In a floorplanning design flow the system is divided into partitions which can be implemented independently. This has the potential to guide the tools towards a better solution, but can also restrict cross-boundary optimizations.

Consider a design of cascaded FIR filters. The implementation produced by Altera's Quartus II 12.1 CAD system using flat compilation is shown in Fig. 1a. Given that each FIR filter is only connected to the preceding and following filters, one would expect them to be well localized. While this is true in many cases, it is clear that the flat compilation process results in significant smearing between instances. In particular, the five most timing critical instances shown in Fig. 1b are stretched out significantly, limiting the achievable clock period.



(a) FIR filter cascade. Each filter has a unique colour.

(b) Cropped view showing the critical paths (highlighted blue) of the five most critical instances.

Fig. 1: Quartus II flat implementation of a FIR filter cascade in a Stratix IV EP4SGX230 device.

In scenarios like this the designer's intuition that each instance can be localized can be used to improve the result. Manually floorplanning a variation of the FIR filter cascade improved the achievable operating frequency from 375 MHz to 417 MHz (+11%). Manual floorplanning was also found to improve frequency in [2], and is promoted by FPGA vendors [3, 4, 5] as a method to address timing closure issues.

Floorplanning is also beneficial for team-based design, allowing design teams, which often already collaborate and partition the RTL design, to extend this partitioning to the physical implementation. This enables independent parallel physical implementation reducing turn-around-time, and also helps reduce coupling between components, potentially reducing the number of design iterations. Floorplanning is also a required step in partial reconfiguration design flows [6, 7], where it is used to define the fixed interface between the static and reconfigurable parts of the design.

Manual floorplanning is a time consuming process, motivating the effort to automate it. While automated floorplanning for ASICs has been well studied (see [8]) the FPGA floorplanning problem is significantly more difficult. FPGA floorplanning tools must handle the discrete and heterogeneous types of prefabricated FPGA resources, unlike ASICs where there is a single continuous resource (silicon area). This requires custom FPGA floorplanning tools.

III. FLOORPLANNING FLOW & PREVIOUS WORK

The design flow we use for floorplanning is shown in Fig. 2. Initially, a flat technology mapped netlist is produced by logic synthesis. The netlist is then partitioned[3], either by an automated tool or by the user. Once partitioned, the netlist is packed into functional blocks (logic, RAM etc.) while respecting partitioning constraints. We perform packing

[1] HETerogeneous Region Implementation System

[2] For instance, placement can place a primitive at any legal location.

[3] Another possibility would be to perform partitioning before logic synthesis.

978-1-4673-9091-0/15/$31.00 © 2015 IEEE

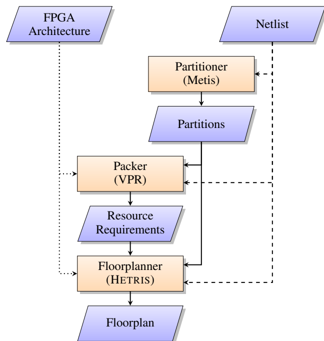
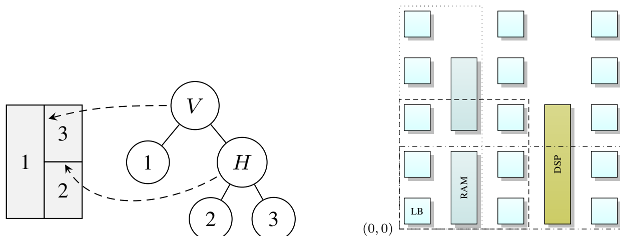


Fig. 2: Floorplanning flow used to evaluate HETRIS.



(a) Slicing tree and relative partition placement. Dashed lines relate internal nodes and cut-lines.

(b) The three realizations in an IRL located at (0, 0) requiring 5 LB and 1 RAM.

Fig. 3: Slicing Tree and IRL example.

before floorplanning so we have accurate partition resource requirements[4]. The floorplanning tool takes as input the target FPGA architecture, netlist connectivity, netlist partitions and partition resource requirements. It then attempts to find a valid floorplan, reporting a solution if found.

Several works have previously addressed FPGA floorplanning. Cheng and Wong [9] present an FPGA floorplanner based on Simulated Annealing (SA) using slicing trees to represent the relative partition positions. A slicing tree is a full binary tree (see Fig. 3a) where leaf nodes represent partitions, and internal nodes represent either vertical (V) or horizontal (H) cuts. Cheng and Wong introduce Irreducible Realization Lists (IRLs) to convert a slicing tree into an exact floorplan. An individual 'realization' is a rectangular region at a particular device location satisfying some resource requirements. A realization is 'irreducible' if it is the area minimal realization for its aspect ratio. An IRL consists of 'irreducible realizations' with various aspect ratios (see Fig. 3b). Cheng and Wong showed how an internal node's IRL can be efficiently calculated from the IRLs of its children. This can be applied recursively to a slicing tree to generate an exact floorplan. Cheng and Wong also use a vertical compaction technique to legalize vertically illegal solutions.

In [10] an ASIC floorplanner is used for floorplanning, followed by network flow based post-processing to re-allocate heterogeneous resources. A greedy back-tracking algorithm is proposed in [11], but suffers from high complexity. [12] presents an interesting multi-layered approach which attempts to minimize wasted resources when partitions are unbalanced. [13] uses techniques similar to [9], but generates slicing trees using a partitioner and performs coarser resource allocation. Floorplanning for partial reconfiguration is considered in [14, 15, 16].

[4]The complex legality requirements of modern FPGA architectures makes it difficult to predict the required resources from only the input netlist.

IV. Algorithm Overview & Run-time Improvements

HETRIS builds upon [9], since it appears to be the most general and robust approach. It uses SA as the outer optimization algorithm operating on slicing trees, and uses IRLs to realize floorplans. At every move, the slicing tree is realized and the resulting floorplan's legality, area and wirelength are evaluated.

One of the key operations is converting from the abstract floorplan representation (e.g. slicing trees) to a concrete floorplan realization with precise locations and dimensions. We propose several enhancements to the IRL algorithm used by Cheng and Wong [9]. More detail is provided in [17].

A. Slicing Tree IRL Evaluation as Dynamic Programming

Although not originally presented as such, Cheng and Wong's IRL-based slicing tree evaluation algorithm can be re-formulated as a case of dynamic programming [18] since:

1) The problem exhibits *optimal substructure*. The IRL at each internal node of the slicing tree is calculated by combining the IRLs of its children.
2) There exist *overlapping subproblems*. Different annealing moves may require calculation of the same IRL.

These insights can be exploited to further optimize run-time.

B. IRL Memoization

The first optimization we propose is to memoize IRLs (subproblems) across SA moves. This avoids re-calculating IRLs multiple times during the anneal[5].

One potential concern about memoizing IRLs is the memory required. In HETRIS, the look-up is implemented as a dynamically sized cache, enabling a space-time trade-off.

C. Lazy IRL Calculation

In Cheng and Wong's work they pre-calculate IRLs for every partition (leaf node in the slicing tree) at every unique location in the FPGA. This requires $O(w_p h_p W_{max} H_{max})$ time, where w_p and h_p are the dimensions of the basic tile while W_{max} and H_{max} are the maximum dimensions of a realization.

Since SA samples only a small part of the solution space, pre-calculating IRLs for every partition at every location is unnecessary. Instead we can extend the memoization procedure to calculate the IRLs of leaf nodes only as required. This 'lazy calculation' avoids calculating unused IRLs. This is particularly relevant for modern FPGA devices which are not tile-able[6].

D. Evaluation

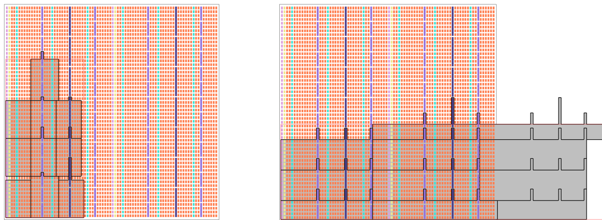
To evaluate the presented run-time improvements, we evaluated the performance of HETRIS by selectively enabling the memoization and lazy evaluation optimizations. Table I illustrates the effectiveness of these optimizations, showing an average 15.6× speed-up. On a per-benchmark basis the best speed-ups (e.g. 31.3× on `des90`) are obtained on small benchmarks, while the speed-up decreases on larger benchmarks (minimum 7.2× on `gsm_switch`) due to the larger portion of time spent on wirelength evaluation. The quality of results (QoR) for all algorithmic variations in Table I are identical, since identical IRLs are calculated.

[5]Cheng and Wong [9] pre-calculated IRLs for leaf nodes, effectively memoizing only the base-case of the recursion.

[6]In this situation $w_p = W$ and $h_p = H$ so the resulting complexity would be $O(W^2 H^2)$, where W and H represent the device width and height — prohibitively expensive for large devices.

TABLE I: Run-time of IRL memoization and lazy calculation optimizations on 17 Titan benchmarks. Each benchmark was partitioned into 32 parts and floorplanned on a tile-able Stratix IV-like architecture. The algorithm in [9] corresponds to 'Exhaustive Memoize Leaves'.

Benchmark	External Net Count	Lazy Memoize All (min)	Lazy Memoize Leaves (min)	Exhaustive Memoize All (min)	Exhaustive Memoize Leaves (min)
gsm_switch	241,048	22.06	44.45	67.59	157.86
sparcT2_core	182,698	17.86	48.47	61.69	154.65
mes_noc	115,606	66.78	212.36	251.83	619.05
minres	112,234	7.63	19.82	41.59	92.39
dart	108,408	13.77	40.87	65.55	155.53
SLAM_spheric	82,370	7.00	22.26	45.44	104.33
denoise	76,377	16.10	52.80	82.86	214.34
cholesky_bdti	74,921	7.42	21.94	47.14	113.23
segmentation	73,086	11.04	37.53	73.35	162.80
sparcT1_core	70,874	5.36	16.20	47.53	102.61
bitonic_mesh	61,110	3.73	6.28	33.88	70.10
openCV	60,981	4.34	10.44	40.56	82.26
stap_qrd	51,755	17.15	58.47	69.02	179.87
des90	37,368	2.38	5.02	36.50	74.55
stereo_vision	35,103	2.34	6.73	33.50	69.64
cholesky_mc	32,408	3.14	13.69	41.85	97.33
neuron	31,365	2.71	11.28	32.96	72.83
GEOMEAN	72,148	7.89	22.74	54.01	123.28
GEOMEAN Speed-Up		15.62×	5.42×	2.28×	1.00×



(a) Homogeneous approximation (b) Heterogeneous-aware (illegal)

Fig. 4: Homogeneous approximation and heterogeneous-aware floorplans for the same slicing tree and benchmark. The homogeneous approximation underestimates the floorplan area by $\sim$2.5× and incorrectly predicts that the slicing tree is legal.

V. Annealer

An equally important component of HETRIS is the outer annealing algorithm.

A. Initial Solution

All SA algorithms require an initial solution. In most previous work, the initial solution is created by solving a simplified version of the heterogeneous floorplanning problem. For instance, Cheng and Wong [9] performed initial floorplanning under a homogeneous approximation which ignores heterogeneous resource requirements. This enabled them to reduce run-time by starting their heterogeneous resource-aware annealer at a lower temperature.

After re-implementing their approach we found the initial solution was no better than starting from an arbitrary initial solution. We believe this is related to the benchmarks and architectures being evaluated. We are using real FPGA circuits to evaluate the floorplanner (Section VIII). In contrast, [9] adapted ASIC floorplanning benchmarks by assuming each partition had heterogeneous resource requirements closely matching the underlying FPGA architecture.

Assuming such a close match is unrealistic. On realistic benchmarks, the homogeneous approximation breaks down (c.f. Figs. 4a and 4b) — reducing the effectiveness of any initial floorplanning. As a result HETRIS constructs an arbitrary initial solution and directly begins resource-aware floorplanning.

B. Annealing Schedule & Moves

The annealing schedule is based on VPR's [19], which adjusts the cooling rate based on the acceptance rate (λ, the fraction of moves accepted), and terminates when the average cost per net becomes a small fraction of the temperature. Also similar to VPR, we perform $O(N^{\frac{4}{3}})$ moves per temperature; where N is the number of partitions to be floorplanned. The annealer uses exchange and rotation moves, which can explore all slicing trees [9].

C. Base Cost Function

An important aspect of any annealer are the cost functions used to evaluate candidate solutions. We define the base cost functions as those used to evaluate the quality of a solution, while cost penalties (Section VII-B) penalize illegality to guide the annealer to a valid solution.

The base cost of a solution S is calculated according to:

$$\textsc{BaseCost}(S) = A_{fac}\frac{\textsc{Area}(S)}{A_{norm}} + E_{fac}\frac{\textsc{ExtWL}(S)}{E_{norm}} + I_{fac}\frac{\textsc{IntWL}(S)}{I_{norm}} \quad (1)$$

where AREA, EXTWL, and INTWL are the same cost components used in [9], and correspond to the area of the floorplan, the centre-to-centre half-perimeter wirelength (HPWL), and an estimate of internal wirelength within a partition, related to region aspect ratio ($\frac{width}{height}$).

The various factors (e.g. A_{fac}) are user adjustable weights used to control the relative importance of the different cost components. Each cost component is normalized by the respective normalization factor (e.g. A_{norm}), whose value is the average of the cost component across the randomized moves made to determine the initial temperature. This ensures each normalized quantity (e.g. $\frac{\textsc{Area}(S)}{A_{norm}}$) is dimensionless with a value of 1.0 on a typical solution.

VI. Solution Space Structure

Given the space of possible solutions, we can view the annealer as traversing a cost surface defined by the base cost function in Eq. (1). Fig. 5 illustrates the explored solution space, allowing us to make several interesting observations.

A. FPGA Architecture and Solution Space Structure

Firstly, solutions are only found at specific discrete locations, creating 'families' of solutions along curves of constant width. Secondly, within each family of solutions, a large number of floorplans with different heights are found, indicating that a floorplan's height is easier to adjust than its width. Thirdly, solutions with small aspect ratios (i.e. tall and narrow, along the left side of Fig. 5) tend to have smaller area.

These characteristics are artifacts of the targeted column-based FPGA architecture. Families of solutions arise since only some region widths will support the required resource types. Consider a region such as the one shown in Fig. 6. Expanding horizontally can be quite disruptive, since it can substantially change both the quantity and type of resources available. In contrast expanding vertically only incrementally changes the quantity of resources available. This makes it more difficult for the annealer to vary a floorplan's width. Finally, tall and narrow floorplans tend to minimize the number of columns with excess/unused resources, minimizing area.

VII. An Adaptive Approach to Legality

Real FPGA devices have a fixed-outline, causing some solutions to be 'illegal', since they fall outside of the device. One approach is to disallow illegal solutions entirely. However, this is difficult to achieve. In particular, without exhaustive evaluation, it is not obvious how to generate a guaranteed legal initial solution or ensure that a potential move will be legal.

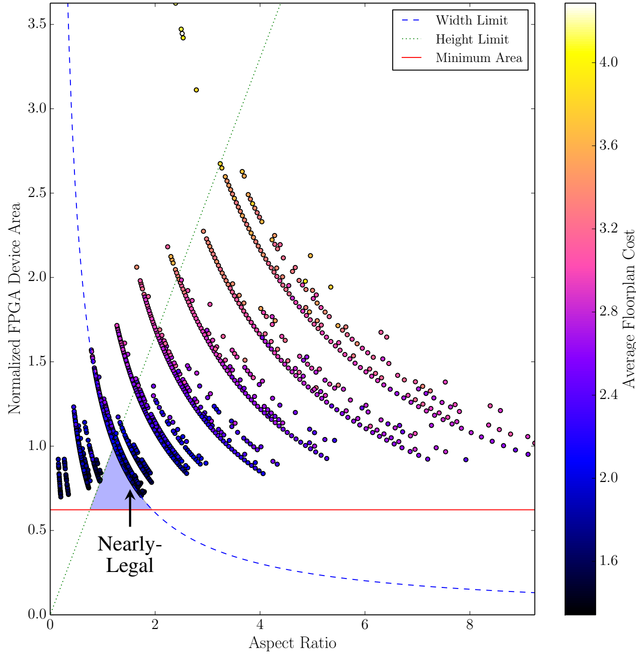


Fig. 5: Base cost surface visualization of explored points in the solution space of the `stereo_vision` benchmark, targeting a tile-able Stratix IV like architecture. Each point corresponds to a specific aspect ratio (x-axis) and area (y-axis). Colour represents solution cost. Hyperbolic curves correspond to solutions with the same width. Diagonal rays correspond to solutions with the same height. An area of 1.0 corresponds to the device size. 'Width Limit' and 'Height Limit' represent the device dimensions. 'Minimum Area' is the area required if partitions are ignored. The shaded triangular-shape denotes the region of legal solutions. No legal solutions were found in this run. The nearly legal solutions clustered along 'Width Limit' are one column wider than the device.

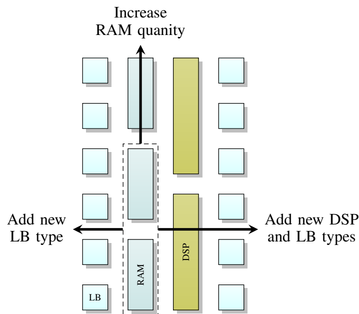


Fig. 6: Different resource types and quantities are available from expanding a region vertically or horizontally.

As a result HETRIS allows illegal solutions. This also has the benefit of allowing the annealer to escape local minima by transitioning through illegal parts of the solution space.

One of the key issues with allowing illegal solutions is ensuring a legal solution is eventually found. To accomplish this, we use a cost penalty to penalize illegal solutions.

A. An Adaptive Approach

One of the considerations when designing a cost penalty is how it should be scaled and evolve during the anneal. The cost penalty must balance two competing factors: ensuring a legal solution is found, and minimizing any impact on the QoR.

It is also desirable for the cost penalty to be robust across a range of FPGA architectures and benchmarks. Rather than expose many tuning parameters, we propose an *adaptive* cost penalty which adjusts[7] to the target architecture and benchmark. This allows the tool to adapt its focus on quality versus legality based on the problem difficulty.

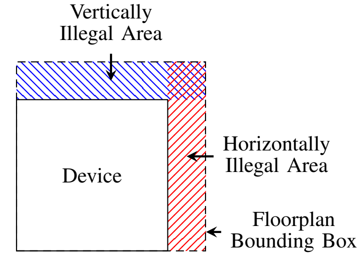


Fig. 7: Example of horizontal and vertical illegal areas.

B. Cost Penalty

Our complete cost function is given by Eq. (2), where H_{fac} and V_{fac} are the current penalty factors, and $\textsc{HorizIll}(S)$ and $\textsc{VertIll}(S)$ measure the horizontally or vertically illegal area of a solution (see Fig. 7).

$$\textsc{Cost}(S) = \textsc{BaseCost}(S) + H_{fac}\frac{\textsc{HorizIll}(S)}{H_{norm}} + V_{fac}\frac{\textsc{VertIll}(S)}{V_{norm}} \quad (2)$$

Interestingly two adaptive parameters, H_{fac} and V_{fac}, are required to produce a robust cost penalty. Using a single parameter causes the annealer to become stuck in an illegal state, since it penalizes vertical and horizontal illegality equally. This prevents transitions from an illegal state in one dimension into a (temporarily) illegal state in the other.

The penalty factors are increased during the anneal based on how successful the annealer is at finding legal solutions. The idea of 'success' is captured by the legal acceptance rate metric (λ_{legal}), the number of legal moves accepted by the annealer. A λ_{legal} of 0 implies no legal solutions have been found, while a value of 1 implies all accepted moves were legal. We also define λ^*_{legal} as the target legal acceptance rate, and λ_{legal_horiz} (λ_{legal_vert}) as the horizontally (vertically) legal acceptance rate.

The value of H_{fac} is updated according to Eq. (3) at the end of each temperature. V_{fac} is updated similarly.

$$H_{fac} = \begin{cases} H_{fac} \cdot P^2_{scale} & \lambda_{legal_horiz}(T) \leq 0.1\lambda^*_{legal} \\ H_{fac} \cdot P_{scale} & 0.1\lambda^*_{legal} < \lambda_{legal_horiz}(T) < \lambda^*_{legal} \\ H_{fac} & \lambda_{legal_horiz}(T) \geq \lambda^*_{legal} \end{cases} \quad (3)$$

P_{scale} is a constant controlling how quickly the penalty factor increases. If the legal acceptance rate is below the target then H_{fac} increases exponentially, otherwise it remains fixed. Empirically we have found P_{scale} in the range 1.005 to 1.2 performs well. Larger values may prematurely freeze the annealer in an illegal state, while smaller values take longer to converge. λ^*_{legal} is typically set to 1.0, ensuring the cost penalties will increase until only legal solutions are accepted.

C. Adjusting the Cooling Rate

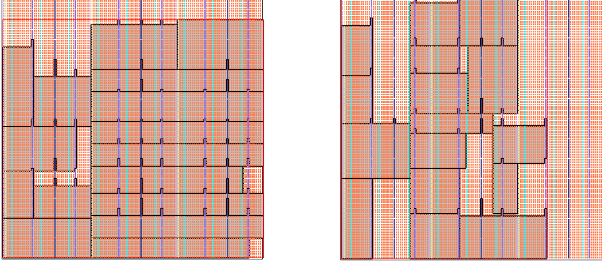
One challenge is ensuring the penalties become effective (of sufficient magnitude to influence the acceptance rate) while the annealer can still hill-climb to a legal solution.

We accomplish this by augmenting the annealing schedule based on the legal acceptance rate. The new temperature update is shown in Algorithm 1. With this cooling schedule the annealer

[7] This is similar to the concept of self-adapting evolutionary algorithms [20], and to the adaptive annealing schedule used in VPR [19].

Algorithm 1 Augmented Adaptive Annealing Schedule

1: **function** UPDATETEMPSTALL(T, λ, λ_{legal}, λ^*_{legal})
2: $T_{new} \leftarrow$ UPDATETEMP(T, λ) ▷ Similar to VPR [19]
3: **if** $0.1 \leq \lambda \leq 0.9$ ▷ Stall only mid-anneal
4: **if** $\lambda_{legal} \leq 0.8 \cdot \lambda^*_{legal}$ ▷ Far from target
5: $\alpha \leftarrow 0.99$
6: $T_{new} \leftarrow T \cdot \alpha$ ▷ Stall
7: **return** T_{new}



(a) A nearly-legal floorplan, only one column wider than the device. (b) A legal floorplan targeting the same device.

Fig. 8: A 'hard' problem for the `stereo_vision` benchmark with 16 partitions, targeting a device only 1.22× larger than minimum.

'stalls' ($\alpha = 0.99$) if the legal acceptance rate is too small[8]. At the beginning ($\lambda > 0.9$) and end ($\lambda < 0.1$) of the anneal the original schedule is used, since stalling would be unproductive.

D. How To Tune A Cost Surface?

For the annealer run in Fig. 5 no legal solution was found, despite exploring many nearly-legal solutions. One of these is shown in Fig. 8a, while a legal solution is shown in Fig. 8b. To transform the illegal floorplan into a legal one, the floorplan needs to be compressed horizontally and expanded vertically.

Adding the illegality terms to the cost function (Eq. (2)) transforms the shape of the cost surface, meaning the annealer is no longer directly optimizing the base cost function[9]. Under this formulation HETRIS is able to find the legal floorplan shown in Fig. 8b. Fig. 9 shows that the cost surface now transitions sharply along the border between legal and illegal solutions; the nearly legal solutions have significantly higher cost. Correspondingly this family of solutions is not explored as extensively. In contrast, the families with legal widths *are* explored more extensively, yielding legal solutions.

Fig. 10 shows the behaviour of the annealer over time. The floorplanner snaps to legal solutions after ∼125 temperatures. Looking at the different cost penalty factors we observe that H_{fac} is more than 3 orders of magnitude larger than V_{fac}. Since their relative magnitude is commensurate with the difficulty of the legality constraint, this confirms our earlier observation that horizontal legality is more difficult to achieve.

It is also interesting to note in Fig. 10 that the EXTWL and INTWL metrics see significant improvement late in the anneal. Despite the floorplan's area being essentially fixed, HETRIS finds new slicing trees with equivalent area but improved region shapes (INTWL) and relative positions (EXTWL).

[8]Note that the annealer does not strictly stall, the temperature always decreases, so the anneal will eventually terminate.

[9]This is similar to barrier methods in continuous optimization, and to STUN a technique to help annealer's escape local minima [21]. In contrast with STUN our approach attempts to guide the annealer towards legal solutions, rather than help it escape local minima.

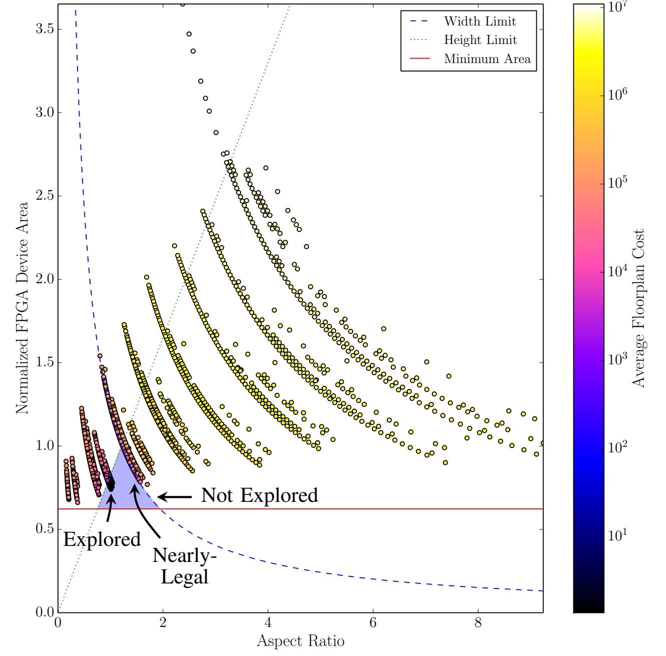


Fig. 9: Cost surface visualization at the end of an anneal with the cost penalty. The benchmark and target FPGA are identical to Fig. 5.

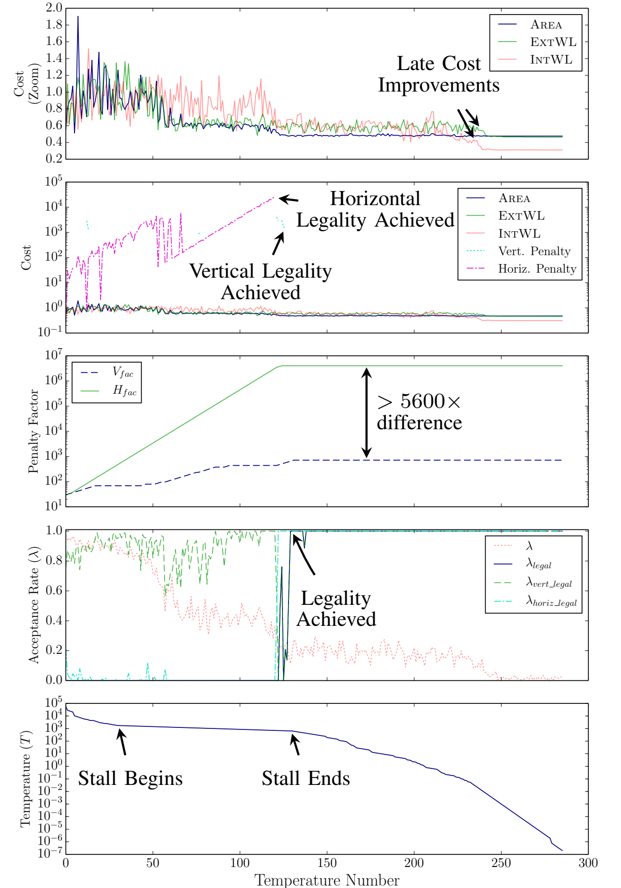


Fig. 10: Annealer statistics as a function of time (number of temperatures) for the `stereo_vision` benchmark.

VIII. FPGA Floorplanning Benchmarks

To evaluate a floorplanning tool, it is important to use realistic benchmarks. This is particularly important since, to the best of our knowledge, *no* previous work on FPGA floorplanning has used real FPGA benchmark designs[10].

The Titan benchmarks are large realistic FPGA benchmarks, suitable for floorplanning [1]. However, they assume a flat compilation flow, and require design partitions to be generated. Partitioning will have a significant impact on any floorplanning design flow, so it is important to make good choices. Given the design dependent nature of partitioning along the logical hierarchy, we focus on physical partitioning, which can be performed by tools such as Metis and hMetis [22, 23].

A. Partitioning Considerations

Automated partitioning tools typically attempt to minimize the (hyper-)graph cut-size, while keeping the different partitions 'well balanced'. The heterogeneous nature of FPGA resources complicates balancing, and precludes using hMetis, as it does not support heterogeneous balance constraints. Metis does support heterogeneous balance constraints, but requires the input netlist to be transformed from a hyper-graph into a simple graph. A variety of netlist transformations have been proposed [24]. We experimentally found a star net model with the inverse net fanout as the edge weights produced good partitions.

Several additional netlist transformations are required to improve Metis' partitioning quality on heterogeneous designs and ensure that partitions are legal. These included preventing Metis from partitioning the primitives representing a single logical RAM, or complex DSP block. Care must also be taken with sparse resources (e.g. PLLs) which may not be balanceable due to their limited quantities. More detail is provided in [17].

IX. Evaluation Methodology

This section describes the methodology used to evaluate HETRIS and empirically investigate the floorplanning problem.

A. Quality of Result Metrics and Comparisons

While we would ideally evaluate the quality of HETRIS by assessing its overall impact on the CAD flow (i.e. post-routing) this falls beyond the scope of this work. Instead, like previous work, we focus on QoR metrics which can be easily measured directly after floorplanning is complete.

It would be desirable to directly compare HETRIS with previous work, but this is not possible. Firstly, there is no consistent set of benchmarks or target architectures used for evaluating FPGA floorplanning algorithms. In particular, the benchmarks used in [9] were never publicly released and are no longer available [25]. Secondly, to the best of our knowledge no previous work has released their floorplanning tools, making direct comparison impossible.

While the algorithms presented in many previous works are important contributions, the heuristic nature of these approaches makes the actual implementation a key component. To help address these issues, we have publicly released the source code for HETRIS along with the full set of floorplanning benchmarks (with partitions) and target architectures used at http://www.eecg.utoronto.ca/~vaughn/software.html.

[10]All previous work has either used synthetically generated benchmarks [11, 12], or various adapted ASIC floorplanning benchmarks [9, 10, 13], which as noted in Section V-A can lead to misleading results.

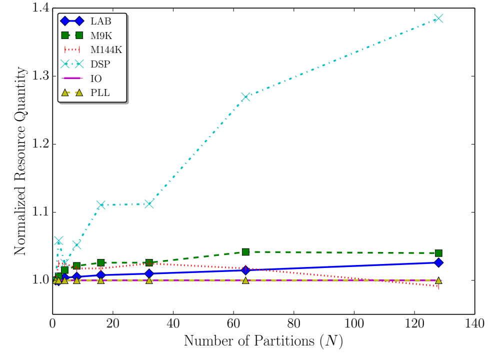


Fig. 11: Resource requirements as a function of partition size. Values are the normalized geometric mean across benchmarks.

B. Design Flow

Fig. 2 illustrates the design flow used to evaluate HETRIS. The initial benchmark netlist is partitioned using Metis. A modified version of VPR 7.0 [26] then packs the netlist into functional blocks (Logic, RAM etc.) while respecting partitions[11]. The resultant packing is used to determine the resource requirements of each partition. Finally, HETRIS floorplans the partitioned netlist onto the targeted FPGA architecture.

C. Target Architecture, Benchmarks and Tool Settings

We target a tile-able version of the Stratix IV architecture in [1]. To make the architecture tile-able, I/Os were placed in columns rather than around the device perimeter, and column spacings were adjusted to follow a repeating pattern[12]. The basic tile of this architecture consists of 336 unique locations, which is larger than the 100 location tile used in [9]. We then floorplan the 17 Titan benchmarks listed in Table I [1].

HETRIS's IRL cache size is left unbounded to ensure all IRLs are memoized. H_{fac} and V_{fac} are initially set to 10, and P_{scale} to 1.10. Unless otherwise noted all base cost component weights (e.g. A_{fac} in Eq. (1)) are set to one. HETRIS was run on systems using Intel Xeon E5-2650 (32nm) processors with 64GB of memory.

X. Floorplanning Evaluation Results

This section performs several different experiments using the methodology described in Section IX.

A. Impact of Netlist Partitioning on Resource Requirements

Partitioning requires that each functional block contain elements only from a single partition, which may increase the required resources. Fig. 11 shows the change in resource requirements as a function of the number of partitions.

Most resources types show only a minimal increase in the required quantity. For example LAB and M9K RAM block requirements increase only 2-3% moving from 1 to 128 partitions. The largest difference is with DSP blocks, which increase by $\sim$38%. The Stratix IV DSP blocks have strict connectivity and legality requirements, making them easy for partitioning to disrupt[13].

[11]This is achieved by modifying VPR's packing algorithm to select only primitives in the same partition as packing candidates for each block.

[12]HETRIS also supports non-tile-able architectures, which are treated as a single tile (see Section X-C). We use a tile-able architecture to be consistent with previous work in [9].

[13]The following device generations (Stratix V and Virtex 7) use simpler DSP blocks which would help alleviate this issue [27, 28].

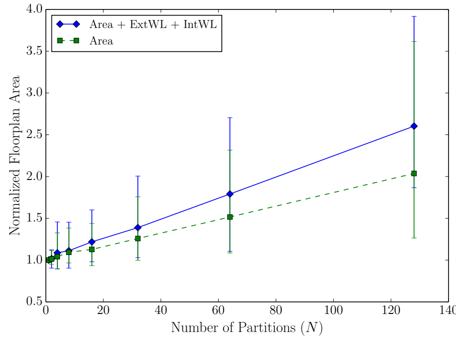


Fig. 12: Geometric mean normalized floorplan area. Error bars denote the range of areas observed across benchmarks.

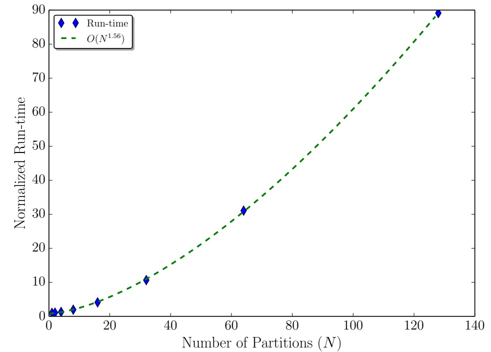


Fig. 13: HETRIS geomean normalized run-time.

B. Floorplanning and the Number of Partitions

The number of partitions used during floorplanning is an important consideration. Smaller, more numerous partitions would improve the speed-up of a flow compiling partitions in parallel, but also may have an impact on area.

Fig. 12 plots the achievable floorplan area against the number of partitions. For the full cost function, the average normalized floorplan area increased from $1.0\times$ to $2.6\times$ moving from 1 to 128 partitions. Running HETRIS in area-driven mode (setting E_{fac} and I_{fac} in Eq. (1) to zero) achieves a smaller increase of $2.0\times$ across the same range.

Partitioning into 6 to 32 partitions appears to be a good choice for typical designs, requiring only a moderate area overhead ($< 1.5\times$) while still exposing significant potential parallelism during the design implementation. However, the best number of partitions is design dependant. Some benchmarks suffer large overheads with only a handful of partitions, while others can easily scale up to 64 or 128 partitions.

Varying the number of partitions also allows us to investigate the scalability of HETRIS. It is important to note that increasing the number of partitions not only increases the size of the floorplanning problem but also increases the number of external nets that must be evaluated by HETRIS. For some benchmarks HETRIS required more memory than was available on the machine[14]. In such cases the benchmarks are excluded from the average. Fig. 13 shows the measured run-time of HETRIS as the number of partitions (N) increases. While the run-time behaviour is super-linear, it maintains a relatively low average complexity of $O(N^{1.56})$. Since we perform $O(N^{1.33})$ moves per temperature (Section V-B) this illustrates the efficacy of the algorithmic optimizations presented in Section IV at reducing the average per-move complexity[15]. For reference, on average at 32 partitions, HETRIS runs in only 7.9 minutes, less than 10% of the average time required by Quartus II to perform full pack, place and route [1].

C. Floorplanning at High Resource Utilization

An important concern is how floorplanning performs at high resource utilizations. To investigate this, we return to the FIR filter cascade design (Section II) which can be scaled to different design sizes. Using this design we can evaluate HETRIS by determining the maximum number of FIR filter instances which can fit on a device. The same experiment can be performed using Altera's Quartus II CAD system with a floorplan generated automatically using 'floating regions', or entered manually. To ensure a fair comparison Quartus II targets a Stratix IV EP4SGX230 device and HETRIS targets a nearly identical device with perimeter I/O (making the architecture non-tileable), and an identical number of LAB, RAM, and DSP resources arranged in the same number of columns and rows.

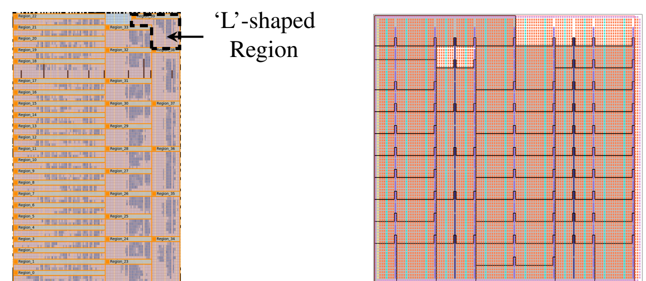
The FIR filter cascade design is limited by the available number of DSP blocks on the device. Table II shows the resource requirements for the different partitioning configurations as well as the maximum number of instances that could (theoretically) fit on the device. The round-off caused by partitioning (since blocks can not be assigned to multiple partitions) can have a significant impact on the maximum number of FIR filter instances that will fit on the device.

TABLE II: Impact of partitioning on FIR Cascade DSP Requirements targeting EP4SGX230 (161 DSP blocks). Each FIR instance requires 26 multipliers, constituting 3.25 DSP blocks.

Partitioning Methodology	Required DSP Blocks per Partition	Effective DSP Blocks per FIR	Number of Partitions on EP4SGX230	Maximum FIR Instances on EP4SGX230
Flat	—	3.25	1	49
1-FIR per Partition	4	4.00	40	40
2-FIR per Partition	7	3.50	23	46

TABLE III: Maximum number of FIRs for which legal floorplans were found in Quartus II and HETRIS. Both the QII partitioned and HETRIS results used 1-FIR per Partition.

Flow	Max FIR Inst.	Time (s)	Note
QII Flat	49	—	
QII Manual FP	40	2,700.0	Required 'L' shaped region
QII Floating Region	37	—	Floorplanning time not reported
HETRIS Default	38	53.9	
HETRIS High Effort + Ignore IntWL	39	135.3	



(a) Manual floorplan requiring an 'L'-shaped region for 40 instances. (b) Floorplan generated by HETRIS for 39 instances.

Fig. 14: Densest manual, and automated floorplans, targeting an EP4SGX230 device. Note that the device aspect ratios are identical in this case, despite being drawn differently by Quartus and HETRIS.

[14] Most of HETRIS's memory is used to cache memoized IRLs across moves. Sizing this cache to the problem would reduce the memory requirements.

[15] In comparison, Cheng and Wong's IRL realization algorithm takes $O(N)$ time [9]. Making the overall complexity of the annealer using their algorithm $O(N \cdot N^{1.33}) = O(N^{2.33})$, substantially larger than what is observed here.

The results of floorplanning with a single FIR per partition are shown in Table III. Flat compilation packs the most instances onto the device, primarily because it doesn't suffer from partitioning round-off effects. Considering partition based approaches, only manual floorplanning is able to fit the maximum number of instances (40). To do so required a non-rectangular 'L' shaped region, highlighted in Fig. 14a. Manual floorplanning required approximately 45 minutes to identify a good floorplan and enter it into the tool. Of the automated methods, Quartus II's floating regions performs worst, fitting only 37 FIR instances onto the device. HETRIS performs better, finding solutions for 38 instances by default and for 39 at a higher effort level and relaxed IntWL (region aspect ratio) cost. The floorplan for 39 FIR instances generated by HETRIS is shown in Fig. 14b. As expected, using automated approaches requires much less time ($\sim 20\times$) than manual floorplanning[16].

TABLE IV: Maximum number of FIRs for which legal floorplans were found in Quartus II and HETRIS, for different partitionings.

Flow	Max. FIR Inst. 1-FIR	Max. FIR Inst. 2-FIR
QII Floating Region	37	40
HETRIS Default	38	44
HETRIS High Effort + Ignore IntWL	39	44

Table IV shows the impact of the different partitioning techniques from Table II. HETRIS is able to pack more FIR instances than Quartus II for both configurations. Overall, the results show that HETRIS is capable of finding legal floorplans even in high resource utilization scenarios, and outperforms Quartus II's floating regions.

XI. CONCLUSION & FUTURE WORK

We have presented HETRIS, an automated FPGA floorplanning tool which can dynamically adapt to arbitrary architectures to generate legal solutions. We also introduced run-time optimizations resulting in an average speed-up of $15.6\times$ compared to previous work. We showed the solution space for modern FPGA architectures is highly non-uniform and developed adaptive annealing techniques to optimize and legalize efficiently in this complex space. We performed the first evaluation of an FPGA floorplanning tool on a set of real-world FPGA benchmarks targeting realistic architectures. These evaluations show that HETRIS is effective at creating optimized FPGA floorplans, and can generate denser floorplans at high resource utilization than Quartus II.

There are many areas for potential future work. How best to size the IRL memoization cache to the problem is an area that requires further study. The bias towards tall and narrow floorplan regions on column-based FPGAs may have implications for floorplanning on interposer based FPGAs. In particular it may be beneficial for the interposer slices to have a similar (tall and narrow) aspect ratio, which is not the case on current devices [29]; however further study is required. As noted in Section X-C, 'L' or 'T' shaped regions may be required to generate legal solutions. This has been studied in ASICs [30], but it is not clear whether these approaches would be effective on heterogeneous FPGAs. Additionally, it would be desirable for such shapes to be automatically selected by the tool without user intervention. It would also be beneficial for HETRIS to optimize additional objectives such as timing. Finally, to prove out its effectiveness, the impact of floorplanning on post-place and route QoR should be investigated.

[16]The FIR design is straightforward to floorplan manually. A more complex design would be significantly more difficult to floorplan manually.

ACKNOWLEDGEMENTS

We would like to thank Jason Luu for his advice on VPR's packing algorithm. This work was supported by Altera, the Government of Ontario, NSERC, the Semiconductor Research Corporation and Texas Instruments. Computations were performed at SciNet [31].

REFERENCES

[1] K. E. Murray *et al.*, "Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD," *TRETS*, vol. 8, no. 2, p. 18, 2015.
[2] D. Capalija and T. Abdelrahman, "A High-Performance Overlay Architecture for Pipelined Execution of Data Flow Graphs," in *FPL*, 2013.
[3] "Lattice Semiconductor Design Floorplanning," Lattice Semiconductor, 2004.
[4] "Best Practices for Incremental Compilation Partitions and Floorplan Assignments," Altera Corp., 2012.
[5] "Floorplanning Methodology Guide," Xilinx Inc., 2012.
[6] "Partial Reconfiguration User Guide," Xilinx Inc., 2013.
[7] "Design Planning for Partial Reconfiguration," Altera Corp., 2013.
[8] T.-C. Chen and Y.-W. Chang, "Floorplanning," in *Electronic Design Automation: Synthesis, Verification and Test*, L.-T. Wang *et al.*, Eds. Morgan Kaufmann, 2009.
[9] L. Cheng and M. D. F. Wong, "Floorplan Design for Multimillion Gate FPGAs," *TCAD*, vol. 25, no. 12, pp. 2795–2805, 2006.
[10] Y. Feng and D. P. Mehta, "Heterogeneous floorplanning for FPGAs," in *Inter. Conf. on VLSI Design*, 2006, p. 6.
[11] J. Yuan *et al.*, "LFF algorithm for heterogeneous FPGA floorplanning," in *ASP-DAC*, 2005, p. 1123.
[12] L. Singhal and E. Bozorgzadeh, "Novel multi-layer floorplanning for Heterogeneous FPGAs," in *FPL*, 2007, pp. 613–616.
[13] P. Banerjee *et al.*, "Fast Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs," *TCAD*, vol. 28, no. 5, pp. 651–661, 2009.
[14] L. Singhal and E. Bozorgzadeh, "Multi-layer Floorplanning on a Sequence of Reconfigurable Designs," in *FPL*, 2006, pp. 1–8.
[15] P. Banerjee *et al.*, "Floorplanning for Partial Reconfiguration in FPGAs," in *Inter. Conf. on VLSI Design*, 2009, pp. 125–130.
[16] K. Vipin and S. Fahmy, "Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration," in *ARC*, 2012, pp. 13–25.
[17] K. E. Murray, "Divide-and-Conquer Techniques for Large Scale FPGA Design," MASc thesis, University of Toronto, 2015.
[18] T. H. Cormen *et al.*, *Introduction to Algorithms*, 2nd ed. Cambridge: MIT Press, 2001.
[19] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL*, 1997, pp. 213–222.
[20] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*, 2nd ed. Springer Science & Business Media, 2004.
[21] W. Wenzel and K. Hamacher, "Stochastic Tunneling Approach for Global Minimization of Complex Potential Energy Landscapes," *Physical Review Letters*, vol. 82, no. 15, pp. 3003–3007, 1999.
[22] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[23] G. Karypis *et al.*, "Multilevel hypergraph partitioning: applications in VLSI domain," *TVLSI*, vol. 7, no. 1, pp. 69–79, 1999.
[24] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration, the VLSI Journal*, vol. 19, no. 1-2, pp. 1–81, 1995.
[25] L. Cheng, Personal Communication, 2014.
[26] J. Luu *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *TRETS*, vol. 7, no. 2, pp. 1–30, 2014.
[27] *Stratix V Device Handbook*, Altera Corp., 2014.
[28] *7 Series DSP48E1 Slice User Guide*, Xilinx Inc., 2014.
[29] K. Saban, "Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency," Xilinx Inc., 2012.
[30] F. Young *et al.*, "On extending slicing floorplan to handle L/T-shaped modules and abutment constraints," *TCAD*, vol. 20, no. 6, pp. 800–807, 2001.
[31] C. Loken *et al.*, "SciNet: Lessons Learned from Building a Power-efficient Top-20 System and Data Centre," *Journal of Physics: Conference Series*, vol. 256, no. 1, 2010.