Resource-Guided FPGA Floorplanning via Quad-Tree-Based NLP Optimization

Ze Chen* and Jordi Cortadella

Universitat Politècnica de Catalunya, Barcelona, Spain
{ze.chen, jordi.cortadella}@upc.edu

Abstract. Field-programmable gate arrays (FPGAs) are widely adopted as accelerators for compute-intensive applications. In modern FPGA flows, floorplanning provides coarse-grained region constraints, but commercial tools offer no automatic support, requiring designers to manually define partition regions that substantially impact placement and routing. This paper presents a fully automatic framework that generates resource-guided floorplanning constraints to improve the clock frequency of FPGA designs. The framework employs a divide-and-conquer strategy with nonlinear optimization for resource-aware region assignment, exploiting FPGA heterogeneity in a device-agnostic manner and producing constraints that integrate into commercial toolflows. The framework has been integrated in Vivado and tested with nine benchmarks and two platforms, improving the maximum clock frequency by 11.4% and 5.8%, respectively, and yielding an additional 4.1% gain with timing-optimization strategies without significant congestion overhead.

Keywords: Field-programmable gate arrays · Floorplanning · Nonlinear optimization · maximum clock frequency

1 Introduction

Due to their reconfigurability, fine-grained parallelism, and energy efficiency, FPGAs have become key accelerators for performance-critical domains such as signal processing and machine learning [6,9]. In modern flows, designers rely on commercial placement and routing engines. However, these tools still depend on manually defined floorplans to guide physical synthesis. Effective floorplanning reduces design iterations and improves subsequent placement and routing.

Research on floorplanning spans multiple device generations. Early works introduced core representations such as slicing trees and sequence pairs, establishing the foundations for layout exploration via stochastic search [7,12]. Homogeneous FPGAs adopted similar formulations, enabling rectangle-based packing and annealing-based optimization [5,10]. With the advent of devices containing BRAM and DSP slices, floorplanning became a resource-aware allocation problem, with methods incorporating column alignment, multi-resource modeling, or analytical formulations to handle heterogeneous footprints [2,11,15].

* Corresponding author.

Still, a gap remains between academic formulations and modern industrial practice. Most prior work assumes rigid, non-overlapping rectangular regions with hard capacity constraints. In contrast, current implementation flows typically interpret floorplanning as soft region guidance: regions may partially overlap, and the placer is allowed to relax local suggestions when necessary to preserve routability or timing closure. At the same time, many real-world designs exhibit deep logical hierarchy and strongly heterogeneous resource requirements. These characteristics motivate formulations that explicitly account for hierarchy and heterogeneous resource footprints, while remaining compatible with soft region semantics and easy to integrate into industrial implementation flows. To the best of our knowledge, unfortunately, there is no literature on resource-aware floorplanning for single-die FPGAs with soft-occupancy constraints. To evaluate the impact of our contributions, we have enhanced a commercial flow with our soft-occupancy constraints and compared results with and without them.

The main contributions of this paper are summarized next:

- The problem of static floorplanning for heterogeneous single-die FPGAs under soft-occupancy constraints is formulated, bridging the gap between rigid academic models and practical industrial flows.
- A scalable divide-and-conquer approach is proposed, that combines hierarchy-aware netlist clustering with mathematical optimization to derive coarse-grained region assignments integrable into commercial CAD flows.
- The framework is architecture-agnostic and easily extensible, enabling deployment across diverse FPGA families and vendor toolchains.

2 Framework overview

The framework is a pre-implementation stage between synthesis and placement. As illustrated in Fig. 1, it consists of four stages: (i) abstraction of the netlist and device layout, (ii) hierarchy-aware netlist clustering and quad-tree construction over the fabric, (iii) hierarchical floorplanning via a nonlinear programming (NLP) model, and (iv) post-processing to generate floorplanning constraints. This flow has been integrated into Vivado [14] to run experiments in Section 4.

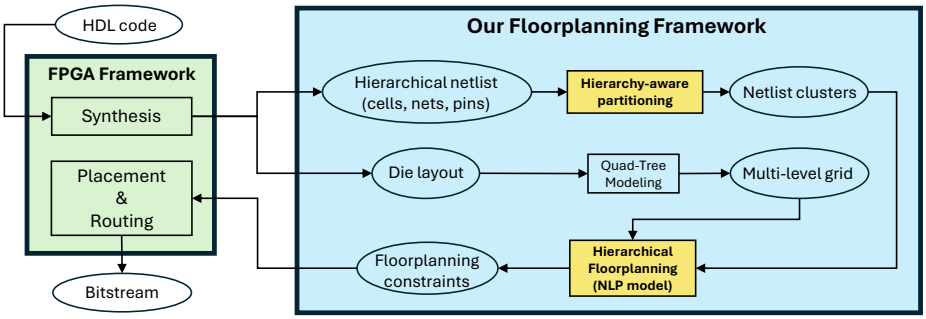


Fig. 1: Flow of the framework

2.1 Hierarchical netlist and die layout abstraction

The starting point of the framework is a synthesized design and the description of the target FPGA architecture. From the synthesis output, we construct a netlist representation that captures cells, pins, nets and the hierarchical organization of cells into modules. In parallel, the device is abstracted as a two-dimensional arrangement of floorplannable units, each labeled with the types and relative amounts of available resources such as logic, memory, and DSP blocks. These units constitute the leaf level of a spatial hierarchy over the fabric. Reserved or forbidden areas of the device (such as configuration columns or dedicated clocking regions) are treated as non-allocatable regions. This abstraction provides a uniform architectural view for the subsequent clustering and floorplanning stages.

2.2 Hierarchy-Aware Partitioning and Quad-Tree Modeling

Given the abstracted netlist and device layout, the second stage builds two coupled hierarchies: a logical hierarchy of netlist clusters and a geometric hierarchy over the fabric.

On the netlist side, the framework performs a bottom-up, hierarchy-aware clustering to derive coarse-grained clusters with strong internal connectivity and limited external communication. Instead of flattening the entire design, the original module hierarchy is used as a guide, so that the clusters preserve module-level structural boundaries while reducing the problem size for optimization.

On the device side, the floorplannable units are recursively grouped into a quad-tree. Neighboring units are aggregated into larger regions, and this process continues until the whole die is represented by a single root node. Each node in the quad-tree represents a region of the heterogeneous fabric together with its available resources. The resulting multi-level grid provides a coarse-to-fine geometric decomposition that defines the search space for the subsequent region-assignment model. The detailed formulations of the clustering and quad-tree construction appear in Section 3.1.

2.3 Hierarchical floorplanning with an NLP model

Given the netlist clusters and the quad-tree over the device fabric, the third stage performs region assignment in a hierarchical manner. Instead of treating each cluster as a rigid rectangle, the framework adopts a soft-occupancy view in which a cluster may occupy several regions with different fractions of its heterogeneous resource demands. At each node of the quad-tree, a small nonlinear programming (NLP) problem determines how the clusters share the resources of the child regions. Capacity constraints ensure that no region is overfilled, while demand constraints ensure that each cluster covers enough resources across the child regions it occupies. The objective function combines two complementary effects: a connectivity-aware term, which encourages strongly connected clusters to stay

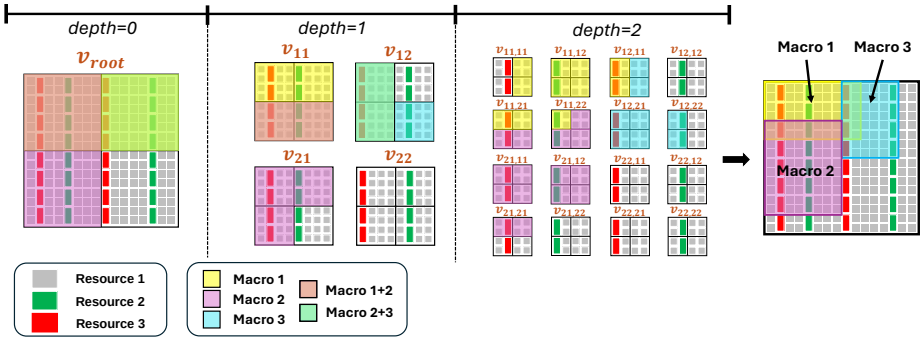


Fig. 2: Recursive refinement along the quad-tree hierarchy

close, and a compactness term, which penalizes overly scattered allocations of a single cluster.

The optimization proceeds top-down along the quad-tree. Starting from the root, which captures a coarse global allocation, each level refines the distribution of occupancies within child regions. This multi-level scheme keeps individual NLP problems small, exploits the hierarchy of both design and device, and yields a set of soft region assignments that are later converted into floorplanning constraints. The full mathematical formulation of the NLP model is given in Section 3.2.

2.4 Floorplan generation and integration

When recursive refinement terminates, each cluster is associated with a set of regions where it has non-zero occupancy. Due to the soft-occupancy formulation, the corresponding footprint typically forms an irregular shape with staircase-like boundaries. Before emitting physical constraints, the framework encapsulates each cluster's occupied area in a minimum-area rectangular bounding box (Fig. 2), and overlapping boxes are allowed because the constraints are used as soft guidances. These boxes are finally translated into floorplanning constraints that can be consumed by commercial FPGA implementation tools.

3 Methodology

This section details the two main components of the proposed framework: hierarchy-aware clustering and quad-tree modeling of the FPGA fabric (Section 3.1), and the quad-tree–based NLP model for hierarchical floorplanning (Section 3.2).

3.1 Netlist Clustering and Fabric Abstraction

(A) Netlist-to-Cluster Modeling Modern FPGA designs contain deeply hierarchical netlists with hundreds of thousands of primitive cells. The clustering stage recursively decomposes this hierarchy into coarse clusters with strong internal connectivity and limited external communication, in a bottom-up manner.

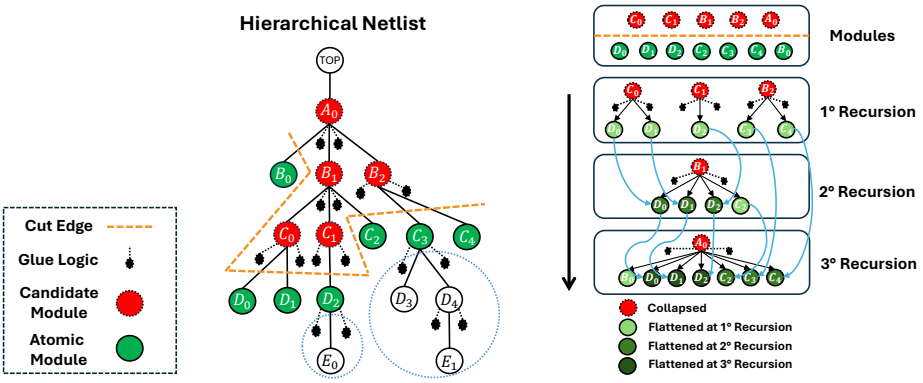


Fig. 3: Illustration of bottom-up recursive netlist partitioning

As illustrated in Fig. 3, the netlist is first scanned to evaluate the aggregated resource demand of each module, i.e., each hierarchical logic block composed of multiple primitive cells. Modules whose resource usage exceeds a threshold are marked as clustering candidates, while smaller or leaf modules (with no children) are treated as atomic and permanently grouped with all their descendants.

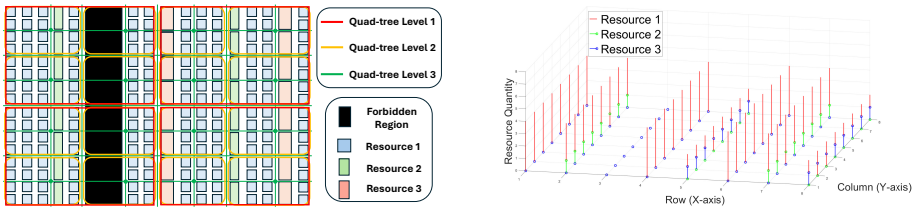
We then traverse the hierarchy bottom-up and process candidates whose children are all atomic. For each such module, we build a local hypergraph capturing its glue logic and child connectivity, partition the glue logic among the children, collapse the parent, and reconnect the children directly to the parent's parent; this recursion continues until the design is flattened into a set of parallel coarse-grained clusters.

Formally, each design segment is modeled as a weighted hypergraph (V, E, ω, c), where each vertex v represents a primitive cell or module with resource demand $c(v)$, and each hyperedge e represents a net with weight $\omega(e)$ proportional to its bit-width. Hypergraph partitioning then produces clusters $C_1, \ldots, C_k$ by minimizing inter-cluster connectivity subject to balance constraints:

$$\min \sum_{e \in E} \omega(e) \cdot \delta(e); \qquad \text{subject to} \quad \frac{c(V_i)}{c(V)} \leq \frac{1}{k}(1+\epsilon), \quad \forall i \in \{1, \ldots, k\}, \quad (1)$$

Here $\delta(e)$ is the number of clusters spanned by hyperedge e, and ϵ is the imbalance tolerance. This hierarchical partitioning flattens the netlist into k coarse-grained clusters $\mathcal{M} = \{m_1, \ldots, m_k\}$, each with aggregated demand $\gamma_{n,t}$ over the type of resource t.

(B) Quad-Tree Abstraction of the FPGA Fabric We now formalize the quad-tree construction sketched in Section 2.2. The heterogeneous FPGA fabric is aligned to a regular grid of base cells whose boundaries coincide with the smallest floorplannable units. Each base cell may contain multiple resource types, and its capacity vector counts how many resources of each type it contains. These cells form the leaf level of the spatial hierarchy. Fig. 4(a) illustrates such



(a) (b)

Fig. 4: (a): Partitioning of heterogeneous FPGA layout (b): 3d footprint of heterogeneous resources of (a)

a subdivision, while Fig. 4(b) highlights the heterogeneous resource distribution across base cells.

Starting from the leaf level, neighboring base cells are recursively grouped into 2×2 blocks to form parent nodes, and the process continues until the entire die is represented by the quad-tree root. Each node j is annotated with a capacity vector $\theta_{j,t}$ equal to the sum of its descendants. Forbidden or reserved regions (e.g., dedicated clocking or configuration columns) are included in the quad-tree by cutting along their vertical boundaries while aligning horizontal cuts with neighboring base cells, while the resulting cells have zero usable capacity.

The resulting quad-tree is the multi-level grid in which the NLP model allocates cluster resources in Section 3.2.

3.2 Quad-Tree-Based NLP Model

Building on the netlist clusters and the quad-tree grids, we now introduce the NLP formulation used for hierarchical node-level resource assignment. For clarity, the main symbols of the model are summarized in Table 1.

Each cluster m_n has heterogeneous resource requirements $\gamma_{n,t}$, and each quad-tree node j provides capacities $\theta_{j,t}$. The continuous occupancy variables $\rho_{j,n,t}$, which represent the fraction of m_n's type-t resource allocated to node j.

Constraints. The first constraint explicitly bounds the occupancy variables:

$$0 \;\leq\; \rho_{j,n,t} \;\leq\; 1, \quad \forall\, j, n, t \tag{2}$$

The second constraint enforces the resource satisfaction for each cluster. Within the current node, the cumulative amount of type-t resources assigned to cluster m_n over all quadrants $\mathcal{J}$ must meet the demand $\gamma_{n,t}$:

$$\sum_{j \in \mathcal{J}} \rho_{j,n,t} \cdot \theta_{j,t} \geq \gamma_{n,t}, \quad \forall n, t \tag{3}$$

To couple allocations between heterogeneous resource types and satisfy the composition of each node, we normalize with respect to the aggregate capacity of that node. Let

$$s_{j,t} = \frac{\theta_{j,t}}{\sum_{t'} \theta_{j,t'}}$$

Symbol	Description
m_n	Cluster n in the clustered netlist, $n \in \{1, \ldots, k\}$
j	Node (quadrant) in the quad-tree grids
t	Resource type (e.g., logic, memory, DSP)
$\gamma_{n,t}$	Demand of cluster m_n for resource type t
$\theta_{j,t}$	Available capacity of type t in node j
$\rho_{j,n,t}$	Fraction of m_n's type-t resource demand assigned to node j
(X_j, Y_j)	Geometric coordinates of the center of node j
(x_n^*, y_n^*)	Center of gravity of cluster m_n
$s_{j,t}$	Share of type-t resource capacity in node j
$\mathcal{R}$	Set of all nodes in the quad-tree
$\mathcal{J}$	Set of child nodes (quadrants) of the current node
E	Set of hyperedges in the clustered netlist
$\omega(e)$	Weight (bit-width) of hyperedge $e \in E$

Table 1: Main notation used in the quad-tree-based NLP model.

denote the share of type-t capacity in node j. The third constraint refers to the normalized cross–type usage bound. In quadrant j, the aggregate occupancy of resource t (summed over all clusters) must not exceed the normalized share $s_{j,t}$:

$$\sum_n \rho_{j,n,t} \;\le\; s_{j,t}, \quad \forall\, j \in \mathcal{J},\ t \tag{4}$$

Connectivity-aware objective. The objective combines a connectivity-aware wirelength term with a compactness term. Since clusters are represented by fractional occupancies of quad-tree nodes rather than rigid shapes, the traditional half-perimeter wirelength (HPWL) is not directly applicable: there is no single bounding box per cluster, and geometry is defined implicitly by the $\rho_{j,n,t}$.

To estimate interconnection cost without requiring explicit rectangular footprints, we adopt a clique-based model that operates on clusters' center of gravity. For every net $e \in E$ connecting $n(e)$ clusters, a complete set of pairwise distances between cluster centers is formed, transforming the multi-pin connection into a fully connected graph (clique). The total interconnection cost is evaluated as

$$WL = \sum_{e \in E} \sum_{p=1}^{n(e)} \sum_{\substack{q=1 \\ q \neq p}}^{n(e)} d(x_p^*, x_q^*) \cdot \frac{2}{n(e)} \cdot \omega(e) \tag{5}$$

where $\omega(e)$ denotes the bit-width of hyperedge e, and $d(\cdot, \cdot)$ is the Manhattan distance between clusters' center of gravity. The L_1 norm can be smoothed, e.g., by replacing $|x_p^* - x_q^*|$ with $\sqrt{(x_p^* - x_q^*)^2 + \epsilon^2}$, to improve numerical robustness.

Centers of gravity. Cluster's center of gravity (x_n^*, y_n^*) reflect the global spatial distribution induced by the occupancy variables. They are defined as capacity-weighted averages over all nodes $\mathcal{R}$:

$$x_n^* = \frac{\sum\limits_{j \in \mathcal{R}} \sum_t \theta_{j,t} \cdot \rho_{j,n,t} \cdot X_j}{\sum\limits_{j \in \mathcal{R}} \sum_t \theta_{j,t} \cdot \rho_{j,n,t}}, \quad y_n^* = \frac{\sum\limits_{j \in \mathcal{R}} \sum_t \theta_{j,t} \cdot \rho_{j,n,t} \cdot Y_j}{\sum\limits_{j \in \mathcal{R}} \sum_t \theta_{j,t} \cdot \rho_{j,n,t}} \tag{6}$$

During the hierarchical optimization, each NLP subproblem only treats the $\rho_{j,n,t}$ associated with the child nodes $j \in \mathcal{J}$ of the current quad-tree node as variables; all other occupancies over $j \in \mathcal{R} \setminus \mathcal{J}$ are inherited from previous levels and remain fixed. This preserves a globally consistent center of gravity while enabling localized refinement within each subproblem.

Dispersion regularization. Minimizing Eq. (5) alone may lead to solutions where clusters are spread thinly across many nodes. To discourage such fragmented allocations, we introduce a dispersion term that measures the spatial coherence of each cluster. Dispersion quantifies how far a cluster's allocated fractions deviate from its own center of gravity:

$$D_{x,n}=\sum_j \sum_t \theta_{j,t} \cdot \sqrt{\rho_{j,n,t}} \cdot (x_n^* - X_j)^2; \quad D_{y,n}=\sum_j \sum_t \theta_{j,t} \cdot \sqrt{\rho_{j,n,t}} \cdot (y_n^* - Y_j)^2 \quad (7)$$

Since $\rho_{j,n,t} \in [0,1]$, the transformation $\sqrt{\rho_{j,n,t}}$ amplifies the contribution of small fractional assignments, making it expensive for a cluster to occupy many nodes.

We aggregate the horizontal and vertical spreads into a single scalar dispersion measure by focusing on the dominant direction. A cluster is considered compact only if both its horizontal and vertical spreads are small; if one direction becomes much larger (e.g., an elongated stripe), the maximum of $(D_{x,n}, D_{y,n})$ reflects this lack of compactness. We therefore define

$$\text{Disp} = \sum_m \max(D_{x,n}, D_{y,n}) \approx \sum_m \frac{D_{x,n} + D_{y,n} + \sqrt{(D_{x,n} - D_{y,n})^2 + 1}}{2} \quad (8)$$

Final objective and hierarchical optimization. The optimization problem at each quad-tree node is a nonlinear program that balances connectivity and dispersion:

$$\min_{\rho_{j,n,t}} \quad \alpha \cdot WL + \beta \cdot \text{Disp} \quad (9)$$

where α and β control the trade-off between wirelength and dispersion.

The NLP is solved recursively along the quad-tree, following the hierarchical scheme outlined in Section 2.3. Starting from the root node, which captures a coarse global allocation, each level refines the distribution of occupancies within its child quadrants while respecting allocations decided at higher levels. This multi-level strategy keeps each subproblem small, and ultimately produces the soft node assignments used to generate floorplanning constraints.

4 Experiments and Evaluation

Implementation and CAD flow. Post-synthesis netlist is exported from AMD Vivado 2024.2.2 and processed by a Python front-end, which builds the clustered netlist and quad-tree grids described in Section 3. Hierarchy-aware clustering uses the hypergraph partitioner *KaHyPar* [8], and each quad-tree subproblem is

modeled in GEKKO [3] and solved with the IPOPT nonlinear optimizer [13]. The resulting per-cluster footprints are written as region constraints (pblocks) into a Vivado `.xdc` file which is either omitted (*baseline flow*) or sourced (*floorplan flow*) before batch implementation so that placement is guided by our framework.

Devices and benchmarks. Experiments are conducted on two FPGA platforms, the AMD KRIA K26 and the Virtex-7 XC7VX415T. Nine benchmark designs are used for evaluation: four CNN accelerators generated by NN2FPGA [4], and five heterogeneous applications from the Koios suite [1]. These benchmarks span a wide range of sizes and resource compositions, including compute-intensive, memory-centric, and irregular workloads. The profiles are summarized in Table 2.

Vivado synthesis strategies. We evaluate both flows with two strategies [14]:

- **Default** strategy: the built-in balanced strategy, which aims to reach timing closure while maintaining moderate compilation time.
- **Performance_ExplorePostRoutePhysOpt(Perf_Opt)**: an aggressive timing-oriented strategy that applies multiple placement and routing algorithms and extensive post-place `phys_opt_design` to maximize timing slack.

4.1 Methodology and Metrics

For each benchmark, implementation strategy, and flow (baseline vs. floorplan), we sweep the target clock period around the expected performance of the design. For each period T, Vivado is run to completion and the worst negative slack (WNS) is collected. The smallest period for which WNS ≥ 0 is denoted $T_{\min}$. The corresponding maximum clock frequency is given by $f_{\max} = 1000/T_{\min}$ (in MHz). In addition to timing closure, we also record several metrics:

- **Routability.** Congestion information is obtained from Vivado's congestion reports. According to UG906 [14], congestion levels above 3 can degrade quality of result (QoR) and those above 6 often prevent timing closure or

Benchmark/Device	**CLB (count / %)**	**DSP (count / %)**	**BRAM (count / %)**
KRIA K26	**14640**	**1248**	**144**
NA1	13005 (88.8%)	764 (61.2%)	51 (35.4%)
NA2	13181 (90.1%)	764 (61.2%)	46 (31.9%)
NA3	12480 (85.2%)	764 (61.2%)	43 (29.9%)
NA4	11415 (78.0%)	764 (61.2%)	57 (39.6%)
Virtex-7 VX415T	**64400**	**2160**	**880**
proxy6	10444 (16.2%)	99 (4.6%)	72 (8.2%)
proxy7	30254 (47.0%)	320 (14.8%)	352 (40.0%)
clstm	39379 (61.2%)	1289 (59.7%)	370 (42.0%)
bwave	14416 (22.4%)	700 (32.4%)	272 (30.9%)
dla	42469 (66.0%)	800 (37.0%)	96 (10.9%)

Table 2: FPGA resource utilization for each benchmark design.

even lead to routing failure. Besides, Vivado typically terminates routing iterations early once timing closure is reached and further improvements are unlikely. We report the number of regions with congestion level > 3, and the number of global routing iterations.
- **P&R Runtime.** We report the runtime of the main implementation stages (placement, post-place `phys_opt_design`, and routing). The runtime of our pre-implementation framework is negligible and is omitted.

4.2 Results and Analysis

Figure 5 shows the maximum clock frequency for all benchmarks under both implementation strategies.

f_{max} under default P&R strategy. Across all nine benchmarks, introducing pblock constraints improves the f_{max} compared to the baseline flow. For the CNN-based NN2FPGA designs on KRIA K26, the average improvement is **25.95 MHz** (**11.36%**); for the Koios benchmarks on Virtex-7, the average gain is **14.98 MHz** (**5.79%**). In every design, the baseline flow fails to meet timing at the f_{max} achieved with pblocks, with worst negative slack deficits ranging from **0.047 ns** to **0.342 ns**. These results show that the proposed floorplanning consistently shifts the timing boundary to a more favorable operating point under a standard, non-aggressive implementation strategy.

f_{max} under Aggressive Timing strategy. With the Perf_Opt strategy, the baseline already achieves higher frequencies than with the default flow, and most benchmarks even exceed the pblock results obtained under the default strategy. Nevertheless, pblocks still push the frequency boundary further: on average we observe an additional average improvement of **10.93 MHz** (**4.06%**) over the corresponding baseline without pblocks. This indicates that global floorplanning and CAD-level timing optimizations are complementary: the former structures the placement space at a coarse granularity, while the latter refines critical paths within the assigned regions.

Congestion and Runtime. Table 3 summarizes two representative implementation points for each benchmark under the `Perf_Opt` strategy: (1) the run corresponding to the maximum clock frequency achievable without pblocks, and (2)

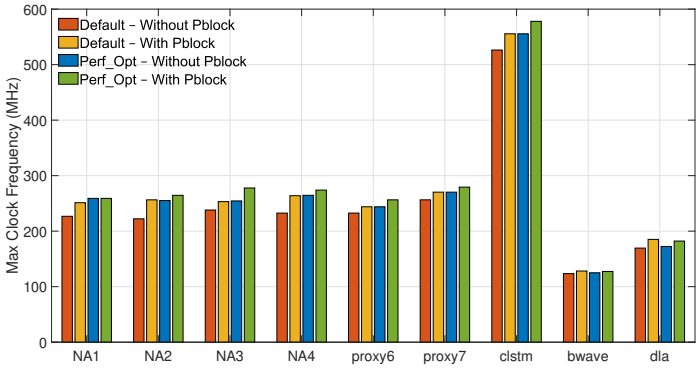


Fig. 5: Impact of pblock constraints on maximum clock frequency.

Benchmark	Placement		PhysOpt		Routing		Total		Cong. Reg.		GR Iter.	
	npb	pb	npb	pb	npb	pb	npb	pb	npb	pb	npb	pb
NA1	8:00	8:21	3:55	4:15	7:25	9:03	19:20	21:39	2	2	4	4
NA2	6:45	6:54	0:18	0:18	7:58	11:32	15:05	18:44	3	3	4	5
NA3	6:40	7:42	0:18	4:23	7:32	9:20	14:30	21:25	1	1	3	8
NA4	5:36	5:57	0:16	2:10	6:22	8:47	12:14	16:54	1	1	5	6
proxy6	2:10	2:20	0:24	0:28	3:51	5:00	6:25	7:48	0	0	4	3
proxy7	6:06	6:58	4:52	6:23	10:49	11:30	21:47	24:51	0	0	5	3
clstm	4:36	5:05	0:22	2:30	7:41	4:15	12:38	11:50	0	0	3	3
bwave	3:00	2:57	1:29	1:39	5:52	5:44	10:11	10:20	0	0	3	3
dla	4:02	4:13	0:15	0:14	3:22	3:26	7:39	7:53	0	0	2	2

Table 3: Comparison of different runtimes (min), number of congestion regions (level > 3), and global routing iterations under `Perf_Opt` strategy with (pb) and without (npb) pblock constraints.

the run corresponding to the maximum clock frequency achievable with pblocks. It provides a direct comparison of runtime and routability at the highest frequency each flow can achieve.

With pblock guidance, implementation runtime typically increases moderately, as placement, physical optimization, and routing explore a more structured, region-constrained search space to meet tighter timing objectives.

From the routability perspective, both congestion level and global routing iterations remain well behaved. For the larger NN2FPGA designs (NA1-NA4), which already exhibit non-negligible congestion in the baseline flow, the pblock-guided flow does not increase the number of congestion regions, and the number of global routing iterations is typically increased or unchanged, reflecting the additional effort required to meet tighter timing objectives in designs that are close to congestion limits. For the smaller Koios benchmarks, congestion levels remain stable, and the number of routing iterations is often reduced or unchanged, consistent with the intuition that, in relatively uncongested designs, coarse region guidance helps the router converge more quickly to a timing-closed solution.

Overall, under an aggressive timing-optimization strategy, pblock guidance achieves a higher frequency at the cost of a modest runtime increase, without introducing additional routing risk or significant implementation overhead.

5 Conclusion

We presented a resource-aware floorplanning framework for single-die FPGAs that integrates hierarchy-aware netlist partitioning with quad-tree–based NLP optimization. The approach can be incorporated into industrial flows. Experiments on nine benchmarks show consistent $f_{\max}$ improvements with modest P&R runtime overhead and without degrading routability, as reflected by stable congestion levels and similar routing-iteration behavior. Future work includes exploring shape-aware netlist modeling, extending the framework to multi-objective formulations that jointly consider timing, routability, and power, and applying the approach to emerging architectures with finer-grained heterogeneity.

Acknowledgements. This work has been supported by funds from the Spanish Agencia Estatal de Investigación under grant PID2020-112581GB-C21 (MOTION), and by the China Scholarship Council (CSC). The authors would also like to thank Luciano Lavagno and Yimamu Yilihamujiang for their helpful discussions and valuable feedback on this work.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Arora, A., et al.: Koios 2.0: Open-source deep learning benchmarks for FPGA architecture and CAD research. IEEE TCAD **42**(11), 3895–3909 (2023)
2. Banerjee, P., Sur-Kolay, S., Bishnu, A.: Floorplanning in modern FPGAs. In: 20th Int. Conf. on VLSI Design held jointly with 6th Int. Conf. on Embedded Systems (VLSID'07). pp. 893–898. IEEE (2007)
3. Beal, L., Hill, D., Martin, R., Hedengren, J.: GEKKO optimization suite. Processes **6**(8), 106 (2018)
4. Bosio, R., Minnella, F., Urso, T., Casu, M.R., Lavagno, L., Lazarescu, M.T., Pasini, P.: NN2FPGA: Optimizing CNN inference on FPGAs with binary integer programming. IEEE TCAD **44**(5), 1807–1818 (2025)
5. Emmert, J.M., Bhatia, D.: A methodology for fast FPGA floorplanning. In: Proc. of the 7th Int. Symp. on FPGAs. pp. 47–56 (1999)
6. Gandhare, S., Karthikeyan, B.: Survey on FPGA architecture and recent applications. In: Proc. ViTECoN. pp. 1–4 (2019)
7. Murata, H., Fujiyoshi, K., Nakatake, S., Kajitani, Y.: VLSI module placement based on rectangle-packing by the sequence-pair. IEEE TCAD **15**(12), 1518–1524 (2002)
8. Schlag, S., Heuer, T., Gottesbüren, L., Akhremtsev, Y., Schulz, C., Sanders, P.: High-quality hypergraph partitioning. ACM J. Exp. Algorithmics **27**, 1–39 (2023)
9. Shawahna, A., Sait, S.M., El-Maleh, A.: FPGA-based accelerators of deep learning networks for learning and classification: A review. IEEE Access **7**, 7823–7859 (2018)
10. Shi, J., Bhatia, D.: Performance driven floorplanning for FPGA based designs. In: Proc. of the 5th Int. Symp. on FPGAs. pp. 112–118 (1997)
11. Singhal, L., Bozorgzadeh, E.: Heterogeneous floorplanner for FPGA. In: 15th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM 2007). pp. 311–312. IEEE (2007)
12. Wong, D., Liu, C.: A new algorithm for floorplan design. In: 23rd ACM/IEEE Design Automation Conference. pp. 101–107 (1986)
13. Wächter, A., Biegler, L.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Program. (106), 25–57 (2006)
14. Xilinx Inc.: Vivado design suite user guide (2024)
15. Yuan, J., Dong, S., Hong, X., Wu, Y.: Lff algorithm for heterogeneous FPGA floorplanning. In: Proc. of the 2005 Asia and South Pacific Design Automation Conf. pp. 1123–1126 (2005)