Chapter 10
Physical Implementation

Abstract In this chapter, physical implementation step of FPGA application design will be investigated, including the well-known EDA steps: packing, placement, and routing. Packing cluster the FPGA atoms together into larger design units; placement assign each design unit to a proper location on the device; routing finds the optimized wire path to connect all these design units.

10.1 Packing

Packing is the first and one of the critical steps when implementing a post-synthesis netlist on a given FPGA, as depicted in Fig. 10.1. Packing algorithm aims to cluster logic primitives, e.g., Look-Up Tables, flip-flops, etc., and efficiently map them to logic blocks in an FPGA. Staying at the most upstream of the implementation flow, the result of packing will profoundly impact the placement, and routing results. Its capability of foreseeing critical paths, placement restriction, and routing congestion can significant improve the overall performance of placement and routing. Therefore, packing algorithms have been intensively researched since the born of FPGA devices.

In this part, we will first define the problem of packing algorithms to solve, and then introduce two well-known types of packing algorithms, and also discuss future trends in this field.

10.1.1 Overview

In sophisticated implementation framework, e.g., the one shown in Fig. 10.2, the input and output data structures of packing algorithms are typically well defined and even standardized (See details in Chap. 3). Figure 10.2 depicts a detailed flow chart for all the existing packing algorithms. A packing algorithm converts a post-synthesis netlist to a clustered netlist, with some optional inputs such as a detailed FPGA device modeling. The clustered netlist is the input of downstream engines, i.e., a placement algorithm.

K. Tu et al., *FPGA EDA*, https://doi.org/10.1007/978-981-99-7755-0_10

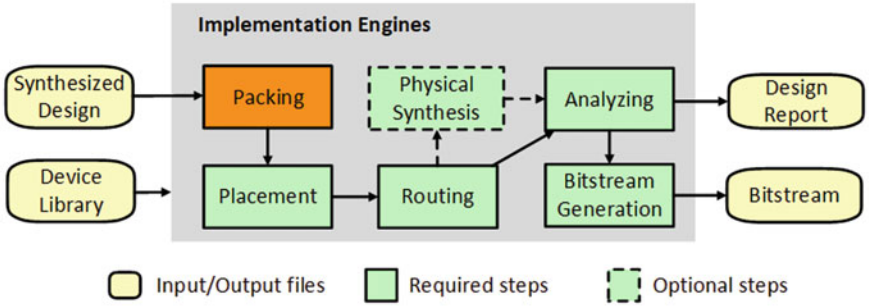


Fig. 10.1 Packing's position in FPGA application EDA flow

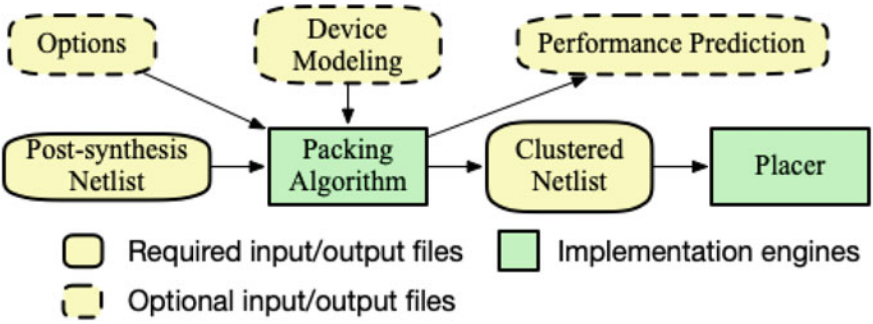


Fig. 10.2 A typical EDA flow for a packing algorithm: input and output data structures

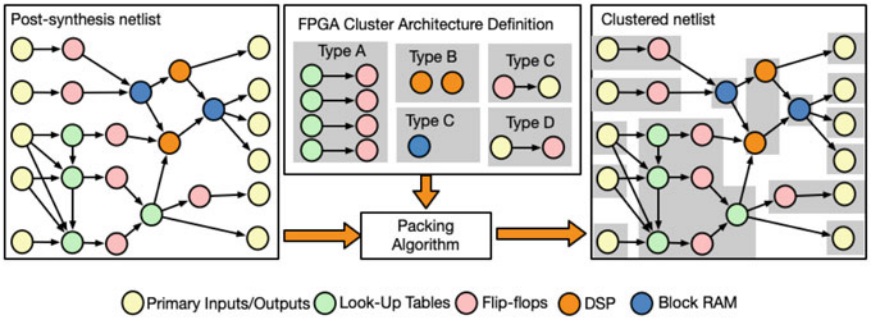


Fig. 10.3 An illustration on the problem definition of FPGA packing algorithms. Each gray box in the clustered netlist denotes a cluster consisting of several primitives

Problem Definition: As an intermediate step between logic synthesis and placements, packing algorithms are designed to cluster and map atom-level logic primitives to dedicated logic resources in programmable blocks, e.g., *Configurable Logic Block* (CLB). (Fig. 10.3) explains the problem definition of packing algorithms through an illustrative example. Packing algorithms require two inputs:

Table 10.1 Comparison on seed-based and partition-based packing algorithms

Metric	Seed-based			Partition-based	Hybrid
Representative tools	TVPack [1, 2]	RPack [3, 4]	AAPack [5, 6]	PPack [7, 8]	HDPack [9]
Support flexible CLB arch.	×	×	✓	×	×
Quality-of-result optimality	Local	Local	Local	Global	Global
Time complexity	Low	Low	Low	High	Medium
Support heterogeneous blocks	×	×	✓	×	×
Open-source	✓	×	✓	×	×

1. A post-synthesis netlist, generated by synthesis tools (See Chap. 9 for details), which consists of a logic network of primitives, such as *Look-Up Tables* (LUTs), *Flip-Flops* (FFs), *Digital Signal Processing* (DSP), *Random Access Memory* (RAM) *etc*. These logic primitives are defined in the technology library of logic synthesis tools.
2. An FPGA cluster architecture description, which describes detailed architectures of programmable blocks in FPGA tiles, such as CLB, DSP, BRAM, and I/O. The technical details include the number of primitives per block, the number input and outputs per block, and also the programmable routing architecture down to pin-to-pin connections. To model the architecture details, packing tools typically build a graph, where nodes represent primitives and edges denote routing resources.

Based on the two inputs, packing algorithms output a clustered netlist which only consists of a number of clusters. Each cluster includes

1. a legal placement of primitives onto logic resources;
2. a legal assignment of nets of the placed primitives to inputs and outputs of clusters;
3. a legal routing of the nets which drives or are driven by the primitives in the programmable block.

Each cluster will be treated as an individual block to be placed for placement algorithms (see details in Sect. 10.2). The nets mapped to inputs and outputs of a cluster will be treated as source and sink nodes for routing algorithms (see details in Sect. 10.3).

In general, to evaluate the quality of packing algorithms, the *Power, Performance, Area* (PPA) at post-routing stage (see Fig. 10.1) are the golden metrics. However, some other metrics are also used to early predict PPA after packing is accomplished:

1. Number of clusters, which indicates the capability of algorithms to group primitives. To maximize the resource utilization of FPGA devices, packing algorithms should result in the fewest number of clusters. Each FPGA device contains a fixed number of programmable blocks for each type (see example in Fig. 10.3. If the

number of clusters exceeds their limits, placement and routing will definitely fail. As a result, a HDL design can not be implemented on the given FPGA device.
2. Number of nets, which indicates the number of interconnections between clusters. The minimize the routing congestion during placement and routing, packing algorithms should absorb as many net as possible into clusters. Any nets remains outside clusters will have to be routed by routing algorithms in downstream stage. FPGA devices contains a limited number of routing resources. A large number of nets may potentially cause overuse of routing resources, leading to failures in routing stage.

Mainstream Algorithms: Depending on the FPGA architecture, packing strategies can be very different. Mainstream packing algorithms can be categorized into three classes: seed-based, partition-based, and a hybrid of the previous two. Table 10.1 compares critical features between the different types of algorithms and their representative tools.

1. The seed-based algorithms are most widely researched and published in the past decade [1–6, 10, 11]. The seed-based algorithms highly rely on a cost function to guide the clustering engine, in order to optimize P.P.A.. The cost function has been intensively studied, which result in various algorithms/tools. Due to the bottom-up nature, the algorithms often hit a local optimal in clustering results. However, thanks to their simplicity and flexibility, the seed-based algorithms are default packing algorithm *AAPack* in the well-known academia FPGA architecture exploration tool *Verilog-to-Routing* [6, 12]. AAPack can now supports highly flexible CLB architectures as well as DSP, BRAM blocks, which are ubiquitous in modern FPGAs devices.
2. The partition-based packing algorithms utilize the graph partition algorithms, e.g, hMetis [13], to produce initial packing results and apply refinement to legalize each cluster. When compared to seed-based algorithms, the partition-based algorithms are more time-consuming (10×) due to the use of graph partitioner [7, 8]. However, partition-based algorithms follow a top-down optimization strategy, which can achieve global optimal results. On average, it can improve P.P.A. by 12% and routability by at 32% when compared to seed-based algorithms.
3. To combine the benefits of seed-based and partition-based algorithms, hybrid algorithms are proposed to perform partitioning as a coarse placement and then annotate the predicted physical location to the cost function of seed-based clustering. On average, it can improve P.P.A. by 7% and routability by at 25% when compared to seed-based algorithms. As the algorithm still rely on seed-based clustering, its timing complexity is similar to the seed-based algorithms with a limited overhead (within 13%).

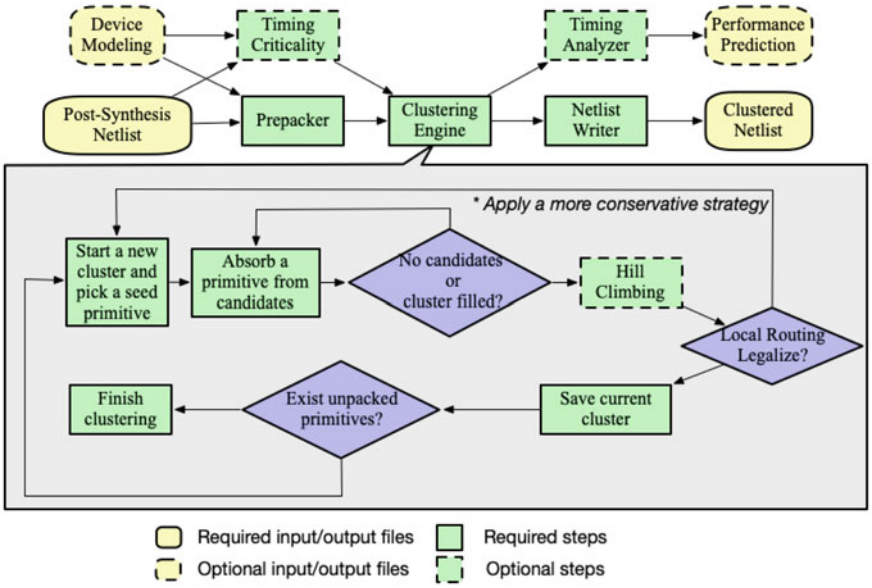


Fig. 10.4 A flow chart to illustrate seed-based packing algorithms

10.1.2 Seed-Based Packing Algorithms

As a mainstream type of packing algorithms, the seed-based packing algorithms have been well studied in the past decades. Even though there are more than ten variants of the seed-based algorithms, the principles are all the same, as illustrated in Fig. 10.2. In this section, we explain the algorithms based on a widely used implementation, i.e., *Architecture-Aware Pack* (AAPack) [6]. Figure 10.4 illustrates the algorithm of AAPack, which is a super set of other seed-based algorithms. The clustering engine of AAPack follows a greedy approach as other seed-based algorithms [1–4]. Clustering is applied to logic blocks, e.g., *Configurable Logic Block* (CLB) *etc.*, in a one-by-one manner, as depicted in Algorithm 1. A new cluster is created with a selected seed, expanded by absorbing primitives through a cost function, and ended when the cluster is fully filled or no more primitives can be absorbed. Each cluster is signed off by a legalizer to guarantee its resource utilization is below a threshold, e.g., 100%. Once a cluster is marked as packed, it will not be revisited by other clusters and no modifications are allowed. As AAPack is designed to be adaptive to various FPGA architectures and offer optimization in different objectives, it contains several unique features (as highlighted by green boxes using dash lines in Fig. 10.4: pre-packing, timing criticality computation, and hill climbing. In the rest of the part, we will present the algorithm details about each technical features.

Pre-pack: To maximize the routability, logic blocks in modern FPGAs typically contains highly flexible routing architectures. However, to efficiently implement

atom_netlist: *Post synthesis netlist*
architecture: *Device modeling showing detailed logic block and routing architectures.*
clustered_netlist: *Outputted netlist which contains clusters only.*
current_cluster: *Current cluster which is open for clustering primitives.*
Function `pack(`*atom_netlist, architecture*`)`**:**
  clustered_netlist = empty;
  while `exist_unpacked_candidates(`*atom_netlist*`)` **do**
    current_cluster = `open_new_cluster(`*atom_netlist, architecture*`)`;
    `try_fill(`*current_cluster, atom_netlist, architecture*`)`;
    clustered_netlist.append(current_cluster);
  end
  return clustered_netlist;
end

Algorithm 1: Seed-based clustering algorithm in AAPack (pseudo-code)

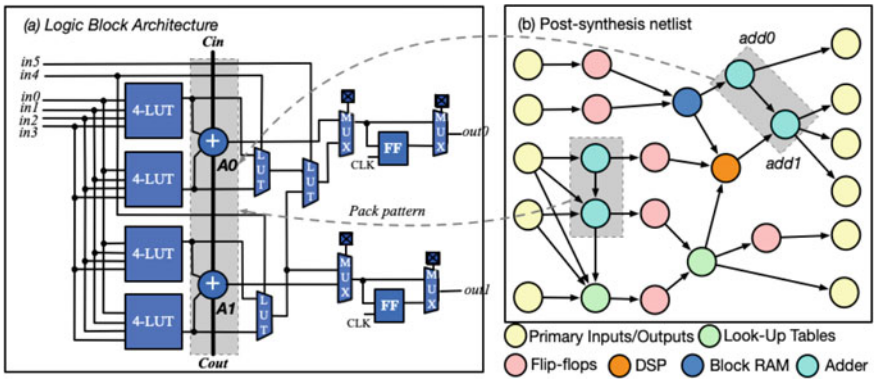


Fig. 10.5 A illustrative example for pre-packing: force packing patterns on post-synthesis netlists based on hard adder chains in a logic block

some frequently used logic functions, such as adder, modern FPGAs also includes a small portion of inflexible routing architectures. These architectures may cause complications in seed-based packers because the packers lack necessary information to map primitives so that the inflexible routing architectures can be satisfied. The most common inflexible routing architecture is on adder chains, as depicted in Fig. 10.5a. The mapping of primitives on the chain has to be exact without any flexibility in location, due to the hard wires in the chain. For example, the two adders `add0` and `add1` in a post-synthesis netlist of Fig. 10.5b, have to be mapped to the logic resource `A0` and `A1` in a cluster, respectively. Therefore, a pre-packing stage is required to extract such patterns of primitives from post-synthesis netlists and force a high priority when mapping to a cluster. As a result, the primitives in a pattern should be added to a cluster only as a group, to avoid potential failures seen in legalizers due to the strong limitation in routing architecture. Algorithm 2 presents the pseudo codes of the pre-packing stage. In the first step, a list of pack patterns

```
<!-- Pack pattern is forced on hard interconnect -->
<direct type="direct2" input="A1.cin" output="A0.cout"/>
 <pack_pattern name="carry_chain" in_port="A0.cout"
out_port="A1.cin"/>
</direct>
```

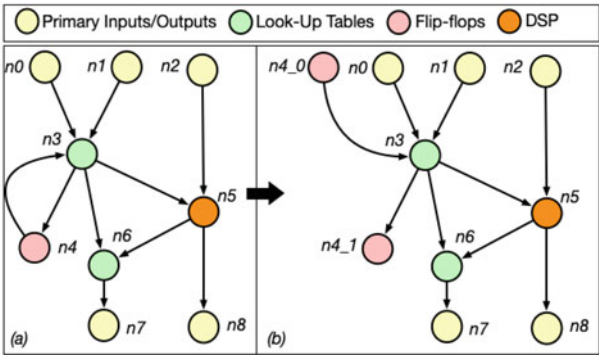
Fig. 10.6 Examples of pack pattern definition in FPGA architecture language

is extracted from the logic block architectures, where the types of primitives that are mappable to the patterns are identified. Note that pre-packing requires users to define pack patterns when crafting their FPGA architectures. Figure 10.6 shows an example of pack pattern for the hard adder chain in Fig. 10.5a, using the *University of Toronto FPGA Architecture Language* (UTFAL) [14]. In the second step, primitives in a post-synthesis netlist are grouped to molecules, which are the smallest unit to be considered/added to a cluster. Note that a molecule may consist of multiple primitives, such as a adder chain, or a single primitive, e.g., a LUT. When traversing the netlist to form molecules, the pre-packer starts with the largest pack pattern and ends with the smallest pack pattern. During each traversal, the pre-packer considers a pack pattern, and try to match parts of the netlist to a pack pattern through a technique similar to the matching process in a standard cell technology mappers [15]. The molecule creation process is greedy to ensure that each primitive can only be assigned to molecule.

atom_netlist: *Post synthesis netlist*
architecture: *Device modeling showing detailed logic block and routing architectures.*
molecule_list: *Group of primitives which are mappable to logic resources.*
current_cluster: *Current cluster which is open for clustering primitives.*
Function `pre-pack(`*atom_netlist, architecture*`)`**:**
    pack_patterns = `extract_pack_patterns(`*architecture*`)`;
    molecule_list = `build_molecule_list(`*atom_netlist, pack_patterns*`)`;
    return pack_patterns, molecule_list;
end

Algorithm 2: Pre-packing algorithm in AAPack (pseudo-code)

Timing Criticality Computation: Timing-driven packing algorithms require timing criticality as a key factor in their cost functions, when select a molecule from candidates to add to a cluster. Therefore, a static timing analysis is run on the post-synthesis netlist, and timing criticality is computed for each net of molecules before clustering stage. Timing parameters of each primitive are annotated from the device modeling, including pin-to-pin delays, setup, and hold time. The timing analyzer predicts the routing delay between molecules based on the segment delays defined in routing architecture of device modeling. For example, AAPack assumes all length-4 wires used to interconnect molecules, which is pessimistic. Through a timing analysis, the slack of each edge is computed, based on which the timing criticality of each edge is achieved. Figure 10.7a shows an example of how a post-synthesis netlist is



(a) An example of timing graph in the context of packer: (a) netlist and (b) timing graph

Cell	T_{delay} (ns)	T_{setup} (ns)
I/O	0.25	-
LUT	1	-
FF	0.5	0.5
DSP	2.5	-
L4 wire	2	-

(b) Cell delays from architecture definition

Fig. 10.7 An example of computing timing criticality in packing

Table 10.2 Timing criticality for each node in Fig. 10.7, where nodes on critical path is highlighted

Node	T_{arrival} (ns)	T_{required} (ns)	Slack	$T_{\text{criticality}}$
$n0$	0.25	0.5	0.25	0.98
$n1$	0.25	0.5	0.25	0.98
$n2$	0.25	6	5.75	0.58
$n3$	**3.5**	**3.5**	**0**	**1**
$n4\_0$	**0.5**	**0.5**	**0**	**1**
$n4\_1$	6	13.25	7.25	0.47
$n5$	**8**	**8**	**0**	**1**
$n6$	**11**	**11**	**0**	**1**
$n7$	**13.25**	**13.25**	**0**	**1**
$n8$	10.25	13.25	3	0.78

converted to a timing graph. Note that a timing path can start from either a primary input or an output of a flip-flop, and end to either a primary output or an input of a flip-flop. Therefore, the node $n4$ is splitted into two nodes $n4\_0$ (as the starting point of a timing path) and $n4\_1$ (as the ending point of a timing path) in the resulting timing graph of Fig. 10.7a. Based on the timing parameters listed in Fig. 10.7b, there is one critical path in the timing graph: $n4\_0 \rightarrow n3 \rightarrow n5 \rightarrow n6 \rightarrow n7$. The critical path delay is $13.75ns$. The timing criticality of each node can be computed by following Eq. 10.1, as detailed in Table 10.2. Note that all the nodes on the critical path have timing criticality of one. By assigning high timing criticality to the nodes on critical paths and close to critical paths, packers can be guided to optimize their timing using cost functions (see details in next part). Typically, seed-based packers cluster these timing critical nodes into the same logic block, in order to minimize the final critical path delay.

$$\begin{aligned} \text{slack}_{\text{node\_}i} &= T_{\text{required, node\_}i} - T_{\text{arrival, node\_}i} \\ \text{timing\_criticality}_{\text{node\_}i} &= 1 - \frac{\text{slack}_{\text{node\_}i}}{\text{crit\_path\_delay}} \end{aligned} \tag{10.1}$$

```
molecule_list: Molecules extracted from post synthesis netlist
architecture: Device modeling showing detailed logic block and routing architectures.
current_cluster: Current cluster which is open for clustering primitives.
Function TryFill(current_cluster, molecule_list, architecture):
    candidates = update_candidate_list_for_cluster(current_cluster,
     molecule_list);
    while is_empty(candidates) or !is_cluster_full(current_cluster) do
        chosen_candidate = pick_molecule_with_highest_gain(current_cluster,
         candidates);
        try_place_molecule_in_cluster(current_cluster, chosen_candidate);
        if cluster_is_legal(current_cluster, architecture) then
            save_cluster_result(current_cluster);
            update_candidate_list_for_cluster(current_cluster,
             molecule_list);
        end
        else
            remove_molecule_from_candidates(chosen_candidate, candidates);
        end
    end
    if !is_cluster_full(current_cluster) then
        try_hill_climb(current_cluster);
    end
end
```

Algorithm 3: Detailed algorithm of the TryFill() function show in Algorithm 1 (pseudo-code)

Clustering engine: Seed-based packers build clusters one at a time, as explained in Algorithm 1. During clustering, molecules, created in the pre-packing stage, are grouped to form a cluster, and mapped to specific locations in the clusters. There are two phases when building a cluster:

1. *Seed selection*: Seed molecule is the first element that is added to a new cluster. Packers pick a molecule based on a cost function, which is considered to bring highest gain to the cluster. The type of gain depends on the objective in optimization, which can be routability-driven or timing-driven, *etc.*. To reach the designated goal, cost functions are built to quantify the gain of each molecule when added to a cluster. Therefore, unclustered molecules can be ranked by the gain, and packers pick a high-ranking molecule as the seed. Table 10.3 lists the seed selection strategy of representative packers. Take the example of VPACK, the molecule `m0` in Fig. 10.8a is selected as the seed when creating a cluster, because it has the most used inputs (= 4).

2. *Cluster filling*: Once a seed is picked, clustering engine fills a new cluster through iterations. Algorithm 3 summarizes the principles of the iterative cluster-filling approach from all the existing seed-based packers. In each iteration, a list of candidate molecules is created based a cost function which quantifies their gain to the cluster. The candidates may be ranked by the highest gain, and the packer can fastly spot the preferred molecule when added it to the cluster. This methodology is similar to the seed selection phase. With a list of the candidates, clustering engine pick a molecule in each iteration and try to place it onto an available location in the cluster. Considering the example in Fig. 10.8a, the molecule `m3` will be placed to the cluster, due to its highest gain than the others `m2` and `m4`. Note that absorbing a candidate with highest gain may not always lead to a success addition. A legalizer is called to ensure that (a) the new member will not exceed the input and output bandwidth of the logic block architecture, (b) when there are local architecture, all the input and output signals can be routed inside the cluster. Take the example in Fig. 10.8a, the molecule `m3` should be legally added to the cluster, as shown in Fig. 10.8b, when the input bandwidth is 6. However, if the input bandwidth is 5, the legalizer will fail when adding the molecule `m3` to the cluster. Instead, the molecule `m2` will be added to the cluster legally. If the legalizer passes, the clustering results, including the placement of the new member and routing traces, are saved and the candidate list will be updated for the next iteration. As a member have been added to the cluster, the gain of unclustered molecules are no longer the same as previous iteration. If the legalizer fails, the candidate will be removed from the list, and will not be considered in future iterations. This phase ends until there are no more candidates or the current cluster is full. To ensure highest gain in each iteration, the selected candidates should have direct connections to the cluster, being either a fan-in or a fanout molecule. As a result, a cluster may not be fully filled due to a limited number of candidates. A hill climbing phase may be called to fill the cluster. Hill climbing aim to increase the resource utilization rate, which considers candidate molecules which have not connections to the cluster. Consider the example in Fig. 10.8c, if there is an input bandwidth of 5, molecule `m3` cannot be added the cluster since it will result in 6 inputs for the cluster. In such case, hill climbing can add an unrelated molecule `m4` to the cluster without violating input bandwidth. Details about hill climbing can be found in [1].

Cost functions: The key factor in seed-based clustering engines is the cost functions, which profoundly impacts the decision in each iteration. Cost functions are considered in both seed selection and cluster-filling phases. Cost functions are design to reflect the affinity between molecules, indicating the gains to absorb a molecule into an existing cluster. A well designed cost function can precisely quantify the gain for each candidate molecule. Provide a rank which clearly distinguish the molecule with highest gain from the others. Therefore, with a sophisticated cost function, a clustering engine can make correct decision in each iteration toward the optimization goal. In contrast, a poor cost function may probably mislead the clustering engines to absorb a improper candidate, being far from the objective, e.g., maximum operating frequency. Depending on the applications, different optimization targets are

Table 10.3 Cost functions in representative packers

Packer	Type	Seed selection phase	Cluster filling phase
VPack [1]	Routability	$\max\{used\_inputs\}$	$\lvert \text{Nets}(M) \bigcap \text{Nets}(C) \rvert$
TVPack [2]	Routability/timing	Same as VPack	$\alpha \cdot TC(M) + (1-\alpha) \cdot \frac{\lvert Nets(M) \bigcap \text{Nets}(C) \rvert}{G}$
RPack [3]	Routability	Same as VPack	$\sum_{i \in \text{Nets}(M)} g(i, \text{Nets}(C), M)$
iRAC [4]	Routability	separation/degree^2	$2 \cdot n \cdot w(M) \cdot (1 + \alpha_{M,C})$
AAPack [5]	Routability/timing	$\alpha \cdot TC + (1-\alpha) \cdot$ used_inputs	$\alpha \cdot \text{TC} + (1-\alpha) \cdot$ connection_gain

M denotes a molecule candidate, while C denotes a cluster
TC denotes the timing criticality in Eq. 10.1

given to clustering engines, resulting in a different choice on cost functions. The two most studied goals are the routability-driven and the timing-driven, respectively. Table 10.3 compares the cost functions used by representative packers in the seed selection phase and cluster filling phase. Early works, such as VPack, RPack and iRac, employ single-objective cost functions and focus on routability optimization [1, 3, 4]. Therefore, the cost functions aim to estimate the number of nets to be absorbed into a cluster. VPack uses the number of common nets between a cluster and a candidate molecule. However, the cost function in VPack is too simplified to optimize routability, especially when there are high-fanout nets. It may mislead packer to prioritize absorbing molecules with common input nets, missing the opportunity to fully absorb low fanout nets. RPack, and iRAC's cost functions focus on fully absorbing nets, and minimize the number of external nets outside a cluster, by leveraging the Rent's rule. For example, in Fig. 10.8a, VPACK will assign the same gain to the candidate molecules `m2` and `m3`, and it may decide to cluster `m3` as depicted in Fig. 10.8b. However, the cost function of RPack and iRAC will assign a higher gain to `m2` than `m3`, because it can reduce more external nets. When comparing Fig. 10.8b, c, the cluster with `m2` contains 5 input nets and 1 output nets, while the cluster with `m3` contains 6 input nets and 1 output nets. Timing-driven packers, such as TVPack and AAPack, introduce the timing criticality into the cost function, as well as a factor α to balance the weights between timing gain and connection gain. The factor α is typically an empirical number achieved by experiments on a set of benchmarks, w.r.t. the best overall performance. When $\alpha = 1$, the cost function is biased to timing critically, resulting in a fully timing-driven clustering strategy. When $\alpha = 0$, the cost function is biased to routability, resulting in a fully connection-driven clustering strategy. We refer interested readers to [1–5] for further details.

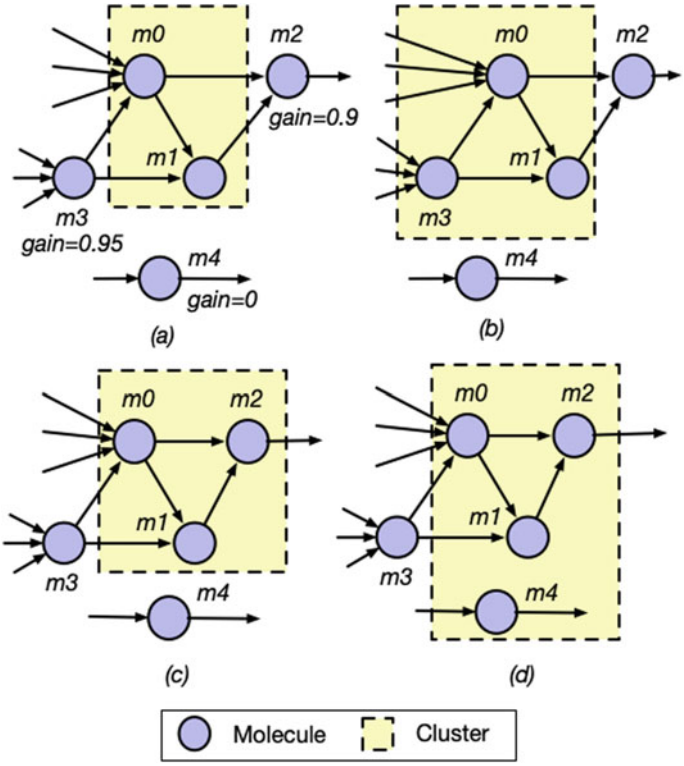


Fig. 10.8 A illustrative example for clustering: **a** an example cluster; **b** absorb the molecule with highest gain; **c** absorb the molecule subject to input bandwidth; **d** hill climbing example

10.1.3 Partition-Based Packing Algorithms

Though seed-based packing algorithms are fast in runtime to produce tight clusters and easy to be tuned for various constraints, they may become stuck in local minima due to the localized greedy strategy. Since clusters are built one by one and clusters cannot be modified once built, it is difficult for seed-based approaches to further optimize QoR from a global view. To compensate the loss, cluster modification, such as *Basic Logic Element* (BLE) swapping can be applied in downstream placement stages [16]. However, such fix is expensive in runtime, which may cause massive changes to clustering results, defeating the initial purpose of packing. Therefore, partition-based packing algorithms are proposed to produce high-quality results through a global optimization approach [7, 8].

To avoid local optimal, partition-based algorithms aim to form clusters with a global view, starting from the first step. Algorithm 4 presents the major steps for partition-based packers [7, 8, 17]. There are three key factors which impacts most on the QoR of the partition-based packers:

```
atom_netlist: Post synthesis netlist
architecture: Device modeling showing detailed logic block and routing architectures.
clustered_netlist: Outputted netlist which contains clusters only.
Function pack(atom_netlist, architecture):
    hypergraph = convert_netlist_to_a_hypergraph(atom_netlist);
    if timing_driven then
        add_weighted_timing_edges_to_hypergraph(hypergraph);
    end
    partitions = kway_partitioner(hypergraph);
    fine_tune_partitions_under_constraints(partitions, architecture);
    clustered_netlist = convert_partition_results_to_clusters(partitions);
    return clustered_netlist;
end
```

Algorithm 4: Partition-based clustering algorithm in PPack/TPPack [7] (pseudo-code)

1. an efficient graph partitioner, which can partition a large graph evely into small groups within a limited amount of runtime. For example, hMetis can produce extremely high-quality bisections of hypergraphs with 100,000 vertices in well under 3 min on an R10000-based SGI workstation and a Pentium Pro-based personal computer [18].
2. a proper modeling of the input netlists `atom_netlist` in hypergraph, which can be accepted by graph partitioners. In particular, the choice of nodes and the weight of edges have profound and direct impacts on the decision of partitioners. For different FPGA architectures, the node and weight build-up may be very different, in order to guide graph partitioner with a clear focus on optimization.
3. rebalancing partitions subject to design constraints. While current state-of-art graph partitioners focus on optimizing connectivity, generated partitions may contain a large of nodes, exceeding the logic capacity of a BLE in FPGA. The rebalancing aims to fixes up these exceptions by moving nodes from oversized partitions to undersized.

In the rest of this part, technical details about each critical step of the partition-based algorithms are presented.

Hypergraph conversion: Hypergraph is a standard and general-purpose graph network which are frequently used by graph partitioners as inputs [19]. Post-synthesis netlists are typically modeled as logic networks, as illustrated in Fig. 10.9a, where each node represents a LUT or a FF or an I/O. When converting such logic network to a hypergraph, a pre-packing step is required, being similar to the seed-based packers (see details in 10.1.2). During pre-packing, LUTs and FFs are paired to become super nodes. Pre-packing not only efficiently reduces the graph size and hence the runtime of graph partitioning, but also avoids the exposure of unnecessary edges which may mislead partitioners. For example, the nodes representing LUTs and FFs `{n0, n1, …, n7}` in Fig. 10.9b are grouped into BLE nodes `{b0, b1, b2, b3}` in Fig. 10.9c. Note that there are tight connections inside each BLE between

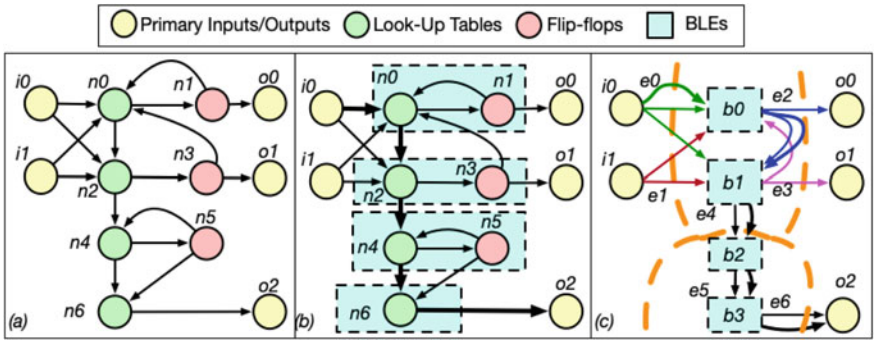


Fig. 10.9 A illustrative example for netlist to hypergraph conversion: **a** an example of input netlist; **b** group nodes into BLEs; **c** hypergraph representation with a potential partitioning

LUTs and FFs. Even these connections are shown in the hypergraph, they may change the partition results with a small probability, as partition algorithms are designed to minimize connectivity. However, as the time complexity of graph partition algorithms is typically high, hiding these details can reduce runtime while have almost no impacts on results. In addition, edges in the logic network are combined to hyperedges between nodes. For example, the edges sharing the same source node `i0` in Fig. 10.9b become a hyperedge `e0` in Fig. 10.9c. We refer interested readers to [19] for concepts of hypergraph.

Timing-driven edge addition: Timing-driven is a critical feature required by modern FPGA design tools. Partitioning-based packers can be made timing-driven by adding a number of extra edges to a hypergraph. This encourages graph partitioner to group the two-end nodes of these edges into the same partition, since the costs of these edges are significantly higher than others. The weight adjustment is done by two step:

1. Identify timing critical edges. The timing criticality or slack of each edge in an input netlist can be computed by the static timing analysis, similar to seed-based packers.
2. For each timing critical edge, add a weighted edge between the two corresponding nodes to the hypergraph. For example, the edges on a critical path (marked by bold lines) in Fig. 10.9b are added to the resulting hypergraph, as show in `{e0, e2, e4, e5, e6}` in Fig. 10.9c.

Note that different packers may have different methods when converting timing criticality to edge weights. For example, for each edge e_i, PPack considers both the pack-accumulated timing weight $pw(e_i)$ and the timing slack slack(e_i):

$$w(e_i) = \alpha \cdot \left(1 - \frac{\text{slack}(e_i)}{\text{slack}_{\max}}\right) + (1-\alpha) \cdot \frac{pw(e_i)^{\beta}}{pw_{\max}} \tag{10.2}$$

where $\text{slack}_{\max}$ and $pw_{\max}$ denote the largest slack and pack-accumulated timing weight among all the edges, respectively. α is the weight factor for slack-based criticality, similar to the weight factor used in TVPack and AAPack shown in Table 10.3. β is an exponent (smaller than 1) to bring all $pw(e_i)$ values to a more comparable level related to $pw_{\max}$ (otherwise, most of them would be near 0). When adding timing edges to a hypergraph, the weight of each edge is scaled to an integer through $[M \cdot w(e_i)]$, where M is the upper bound. In practice, the number of timing edges to be added may be limited by an upper bound p (a ratio on the total number of edges), because it may mislead the partitioner by forcing a strong bias on timing while have no considerations on routability. The parameter set α, β, M, p is empirically obtain through experiments on a benchmark suite. For example, α, β, M, $p = 0, 0.25, 6, 20\%$ is reported by [7] based on the best practice considering the MCNC-20 benchmark suite [20].

Graph Partitioning: Circuit partitioning results are achieved by running a k-way graph partitioner, e.g., hMetis [18, 21]. Even though these modern graph partitioners have been well optimized to produce high-quality results in a reasonable runtime, it may not handle high-fanout nets well. In practice, some high-fanout nets, e.g., fanout > 100 is considered in PPack, are removed in hypergraph without affecting routability. To further reduce runtime and manipulate partitioners, a recursive approach is proposed to apply bipartitioning recursively [8]. We refer interested readers to [21] for details about graph partitioning techniques.

Partition refinement: In this step, partitions are refined to ensure they fit FPGA architecture w.r.t. design constraints, such as logic capacity, routability. For example, some partitions from the k-way partitioner may contain more LUTs or FFs than the number of logic resources in a CLB, since these constraints are difficult to force on partitioners. Refinement is applied by incrementally updating partitions. Firstly, all the nodes in the oversized partitions are identified. For each node, the best target partition is determined by computing the gains of merging the node to candidate partitions. The gain function varies from one packer to another. For example, one can use the attraction function as seed-based packers [17]. Once the best target partition is found, the node is moved and the candidate node pool is updated. Such strategy repeats until there are no oversized partitions. To avoid high runtime when there are a large number of candidate partitions, the refinement may be limited to a small range of partitioners, which are considered to be close in hierarchy [8].

Clustered netlist conversion: Refined partitions are converted to clusters one by one. For example, nodes representing BLEs are recoverd to LUTs and FFs, while hidden connections are restored. This is a straightforward process as each partition is already legalized during refinement.

10.1.4 Summary and Trends

Packing has been intensively studied in past decades with several mainstream algorithms proposed. All these algorithms share the same objective: to properly group

logic primitives into clusters, so that the number of external nets are minimized and critical path delay is reduced. However, to evaluate the quality of results, packing algorithms cannot be simply judged by the number of external nets and a predicted critical path after packing. Results of packing algorithms are evaluated through a complete design flow, i.e., placement and routing followed by an accurate timing analysis.

Seed-based packers are easy to implement due to their open-source availability and simple nature. Seed-based packers have been extended to support heterogeneous blocks, such as DSPs and BRAMs, while partition-based packers are currently limited to LUT and FFs only. Partition-based packers outperforms seed-based packers on MCNC benchmarks on routability (37% smaller in minimum routable channel width) and timing (12% smaller in critical path delays). But partition-based packers are 10× slower than seed-based packers even for small benchmarks, e.g., MCNC, whose number of nodes are no more than 8 k. High-quality results of partition-based packers can reduce the runtime of placement and routing by 34% on average. However, modern FPGA designs may contain millions of LUTs, which still challenges partition-based algorithms due to their high runtime cost. Choice of packing algorithms also strongly depend on the considered FPGA architectures. Seed-based packers can be tuned through pre-packing and cost functions to fit some dedicated FPGA architectures [22, 23]. Partition-based packers are suitable for FPGA architectures without input bandwidth on clusters, which can efficiently reduce the workloads when rebalancing partitions.

In future, packing algorithms may combine the advantages of seed-based and partition-based (see Table 10.1), while avoid the local optimal and long runtime. For example, the HDPack investigates the use of partitioners for a coarse clustering, and then use seed-based algorithms to perform fast and exact clustering.

10.2 Placement

10.2.1 Overview

FPGA placement is a vital process that having a synthesized design netlist (in terms of clusters/molecules/atoms) to be assigned into physical locations under design constraints (Fig. 10.10).

Good placement is extremely important, because even if the circuit is legally implemented, a poor placement could still lead to a low maximum operating speed or a high power consumption (Fig. 10.11).

Problem Definition: Finding a good placement solution is challenging. A modern commercial FPGA can contain over 2,000,000 function units, exhaustive evaluation is therefore impossible. Besides, not every placement solution is legal, it must under some constraints:

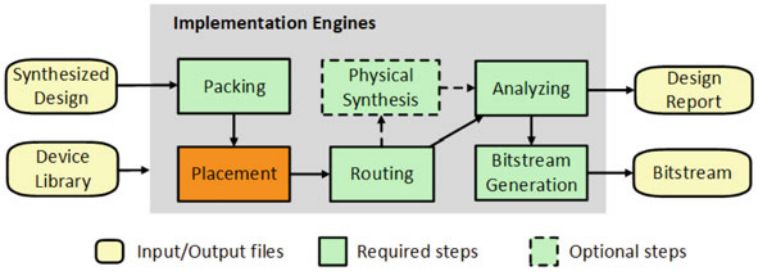


Fig. 10.10 Placement's position in FPGA application EDA flow

Fig. 10.11 FPGA placement matters

1. Accommodation legality limit
   Generic logic cluster must be placed in a generic logic tile (GLT) location, and a memory cluster must be placed in a memory tile (MMT) location, etc. Placement for FPGAs is actually a slot assignment problem.
2. Specific group limit
   Such as arithmetic logic units forming a carry chain must be placed adjacent to each other in the sequence required by the carry structure.
3. Routing congestion limit
   The circumstance that the interconnect exceeds the fabricated wiring capacity in some part of the FPGA must be avoided.

Mainstream Algorithms: Placement engines can be categorized from different perspectives:

1. By algorithmic logic
   Partition-based techniques [24–26] use divide-and-conquer techniques to recursively partition a circuit and induce ever-smaller placement problems. These techniques can achieve a short runtime, but they suffer from low QoR with large-scale designs and therefore hardly adopted by industry. Annealing and analytical techniques, which will be further discussed in the following sections, are the most popular engines both for academia and industry. In fact, modern placement solutions usually apply them in combination to achieve better results. For example, Intel Quartus placer [27] applies analytical method to determine the an initial placement and then uses annealing method to fine tune it.

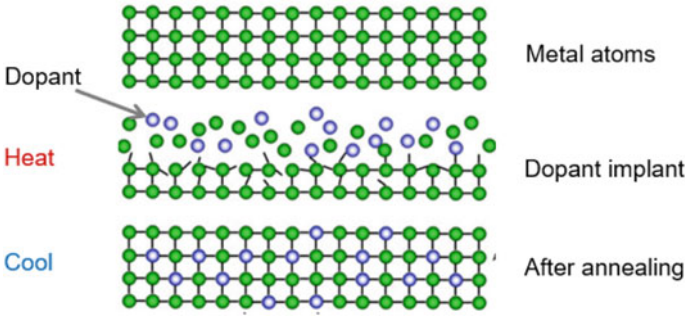


Fig. 10.12 Thermodynamics of physical annealing

2. By optimization objectives
   There are plenty of optimization objectives for placement, for example, wirelength-driven placement minimizes the required wiring; routability-driven placement trying to avoid congestion before routing; timing-driven placement maximizes the circuit speed; power-aware placement takes power consumption into consideration, etc. An excellent placement engine would always balance well among these objectives.
3. By computation acceleration
   In order to reduce compilation time while maintaining quality of results, placement computations can be accelerated either by parallelism or by AI.
4. By targeted architecture
   For other special modern FPGA architecture features (such as advanced package (2.5D/3D), complexed clocking), there are additional placement considerations to adapt to them.

10.2.2 Annealing Placement Algorithms

Physical annealing is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. It occurs by the diffusion of atoms within a solid material, so that the material progresses toward its equilibrium state (Fig. 10.12).

Physical annealing for metal procedures (Fig. 10.13):

Stage1: Heating

Take a metal and heat it to a high temperature. Give it an initial temperature, and atoms transit to high energy states.

Stage2: Cooling

Allow it to cool slowly, metal is annealed to a low temperature. Atoms slowly move to low energy states during the temperature drop.

Higher the initial temperature, slower the cooling, the tougher the metal becomes (Fig. 10.14).

Fig. 10.13 Operation of physical annealing

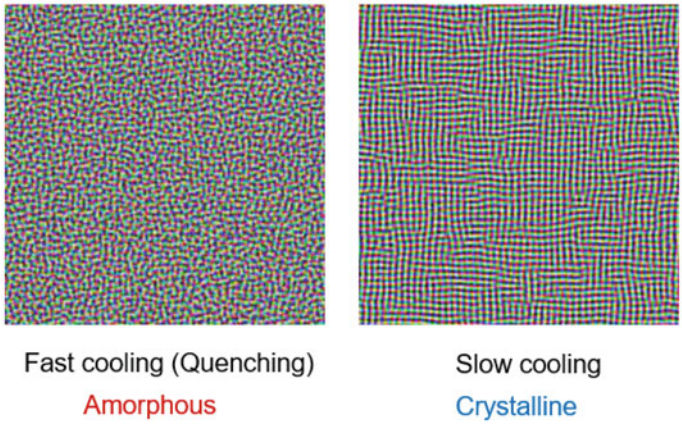


Fig. 10.14 Cooling speed is important in physical annealing

In the field of FPGA EDA, simulated annealing is a probabilistic technique that mimics the physical annealing process, for approximating the global optimum of a given function in a large search space. It was first introduced in 1953 by Metropolis et al. [28] and for solving combinatorial minimization problems and NP complete problem.

Simulated annealing for FPGA placement procedures are (Fig. 10.15):

Stage1: Starts with an initial state and temperature (analogy from thermodynamics).

Stage 2: Randomly changing the state, create a new state.

Stage 3: Compare energies of the new state and the current state, if the new state has less energy, or its probability function $e^{-\Delta C/T}$ is less than a random value (ΔC refers to energy change and T refers to the current temperature), Accept the new state; for circumstances not above, Reject the new state.

```
P = InitialPlacement ();
T = InitialTemperature ();
while (ExitCriterion () == False) do
    while (InnerLoopCriterion () == False) do
        P_new = PerturbPlacementViaMove (P);
        ΔC = Cost(P_new) - Cost(P);
        r = random (0,1);
        if (r < e^(-ΔC/T)) then
            P = P_new;
        end
    end
    T = UpdateTemp (T);
end
```

Algorithm 5: Generic simulated annealing placement algorithm (pseudo-code)

The criteria of simulated annealing placement for FPGA application design includes:

1. Placement Schedule (Annealing)—Defines how to explore the solution space, for example, the starting and ending temperature (the temperature controls the likelihood of accepting moves that make the solution worse); moving strategy; the rate at which the temperature is decreased; the exit criterion for terminating the anneal; the number of moves attempted at each temperature; the method by which potential moves are generated, etc.
2. Cost Function—Energy status of a certain temperature. It is used to evaluate the impact of each proposed move based on the desired optimization objectives.
3. Computation Time—Placement has always been a time-consuming stage, especially for complex FPGA architectures. Research work [29] shows that commercial simulated annealing placer in Quartus II takes about 49% of the compilation time for Titan benchmarks. Reducing computation time is also being intensively studied by both academia and industry.

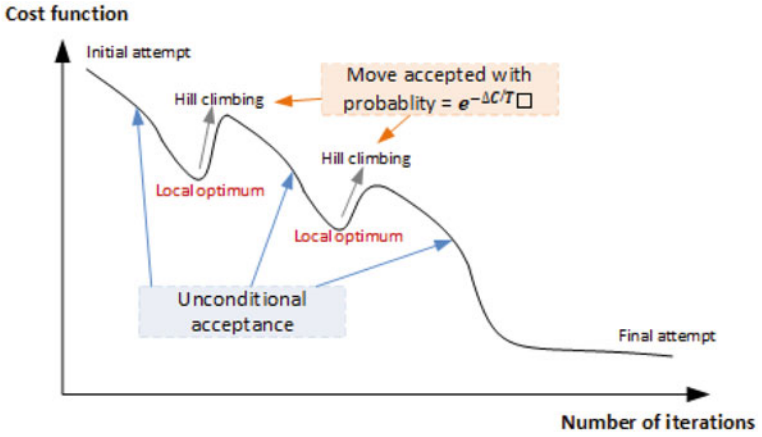


Fig. 10.15 Iteration process of simulated annealing

Placement Schedule (Annealing)

VTR's original VPlace [1] is a famous pioneer work of FPGA placement, according to the algorithm, all the criteria are parameterized (therefore adaptive). In every update, VTR optimize the annealing schedule on the basis of previous version. For example in the newer VTR8 [30], move region limit is optimized by compressed move grid to avoid the situation that preventing a sparse block from moving between columns.

Moving strategy is a researching hot spot in this field. Instead of random move in the original VTR, direct move calculates the effectiveness of each move and make future moves according to this effectiveness [31, 32]. Intel Quartus placer also uses directed moves, but no details has been published [33].

Cost Function

1. Wirelength
   Wirelength, often measured in Half Perimeter WireLength (HPWL), is the most important objective that can not be neglected.
   Bounding box (the smallest rectangle to encloses all the terminals of a net) is introduced to estimate the wiring cost (Fig. 10.16) and the cost function (HPWL of the bounding box) is defined as (Eq. 10.3). bb_x and bb_y are the x- and y-directed span of the bounding box that just encloses all the terminals of the net (Fig. 10.16). With this cost function, place engine could minimize the total HPWL across all considered nets.

$$\text{WireCost} = \sum_{i=1}^{\text{num\_nets}} [bb_x(i) + bb_y(i)] \tag{10.3}$$

   VPR's VPlace [1] uses linear congestion cost function to take routability into consideration (Eq. 10.4).

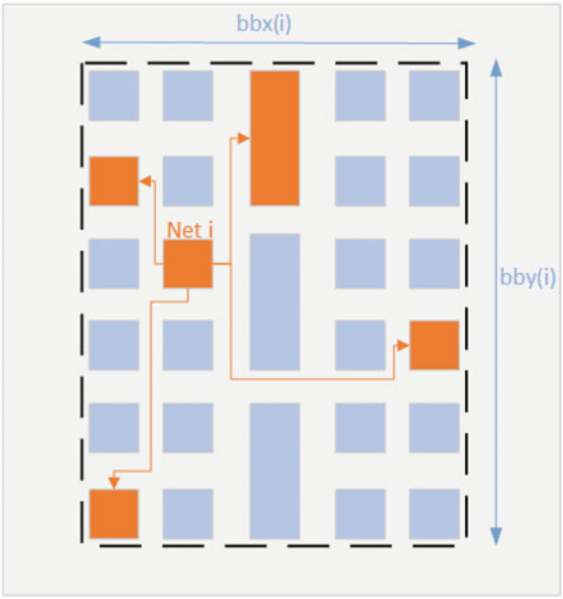


Fig. 10.16 Bounding box is used to estimate the wiring cost

$$\text{WireCost} = \sum_{i=1}^{\text{num\_nets}} q(i) \left[\frac{bb_x(i)}{[C_{\text{av},x}(i)]^{\beta}} + \frac{bb_y(i)}{[C_{\text{av},y}(i)]^{\beta}} \right] \tag{10.4}$$

where $q(i)$ is fanout-based correction factor and linearly increases to help correct the underestimated wiring. $C_{\text{av},x}(i)$ and $C_{\text{av},y}(i)$ are the average channel capacities (in tracks) in the x and y directions respectively, over the bounding box of net i. β allows the relative cost of using narrow and wide channels to be adjusted. The larger value of β, the more wiring in narrow channels is penalized relative to wiring in wider channels.

2. Timing

   The timing cost function can be loosely classified as net-based [34–38], path-based [39, 40] or a hybrid of the two [41, 42]. Net-based cost function usually transform timing to net weights, while the path-based cost function try to represent the timing of critical paths directly. Using path-based cost function generally has more accurate timing view and control, but it suffers from poor scalability and high complexity; using the net-based cost function is very suitable for large chip design such as FPGAs since it has relatively low computational complexity and high flexibility.

   T-VPlace [36] is the timing-driven version of VPlace, it improves the cost function by adding net-based timing considerations (Eq. 10.5).

$$\text{TimingCost} = \sum_{j=1}^{\text{num\_nets}} \text{Criticality}(j) \cdot \text{Delay}(j) \tag{10.5}$$

where $\text{Delay}(j)$ is the delay value of the edge joining node j to node i. For quick estimation, it computes the delay between two blocks as a function only of the distance $(\delta x, \delta y)$. Before that, the placer employs router to determine the delay between two blocks that are $(\delta x, \delta y)$ distance apart, and record it in the delay look up table at location index $(\delta x, \delta y)$. $\text{Criticality}(j)$ (Eq. 10.6) of connection j in the design is determined by periodic timing analysis.

$$\text{Criticality}(j) = 1 - \frac{\text{Slack}(j)}{\text{Delay}_{\max}} \tag{10.6}$$

where $\text{Delay}_{\max}$ is the critical path delay (maximum arrival time of all sinks in the circuit), and $\text{Slack}(j)$ is the amount of delay that can be added to the connection j without increasing the critical path delay.

3. Power
   The power cost function is dependent on the switching activity of the hardware resources of FPGA.
   Lamoureaux [43] modified VPR's cost function by adding a power cost (Eq. 10.7):

$$\text{PowerCost} = \sum_{i=1}^{\text{num\_nets}} q(i)[bb_x(i) + bb_y(i)] \cdot \text{activity}(i) \tag{10.7}$$

where $\text{activity}(i)$ represents the average number of times net i transitions per second.

Total cost function often trades off these objectives above. If the trade-off variable λ determines how much weight to give timing cost, variable γ determines how much weight to give power cost, then the total cost could be described as (10.8).

$$\text{WeightedCost} = \lambda\text{TimingCost} + \gamma\text{PowerCost} + (1 - \lambda - \gamma)\text{WiringCost} \tag{10.8}$$

Computation Time

1. Parallel acceleration
   Distribute placement moves among multiple computation engines to be evaluated in parallel is widely used to shorten annealing time. However, this could lead to conflicts if multiple computational engines accept moves that affect the same design units or nets (termed as "collision") and nondeterministic or serial nonequivalent results.
   Parallel solutions address the problems above and can be inspected from different perspective:

a. In terms of algorithmic logic (Software)
Independent set identifying—Find independent (non-colliding) set of moves and process them all in parallel. The Intel Quartus placer [33] is a representative work in this field, it speculatively evaluates moves in parallel and uses a dependency checker to detect collisions.
Relevant set partitioning—Assign each computation core a partition in the placement area such that different computation unit's moves will not interfere with each other. An early parallel implementation for standard cell placement [44] uses this method to avoid collision.

b. In terms of computation architecture (Hardware)
Scalar architecture (CPU)—Modern CPU usually has multiple cores, and each core has numbers of threads. Using multiple CPU cores or threads as computation engines for annealing placement has been intensively studied [33, 45–48].
Vector architecture (GPU)—Streaming multiprocessor (SMP) is the computation engine of GPU and can be used as annealing placement computation engine to obtain deterministic result. Related works [49, 50] have achieved decent speedups over multi-threaded CPU implementations.
Spatial architecture (FPGA)—In [51], systolic array of processing element (PE) in an FPGA is used as the computation engine. It accelerates the annealing placer and achieves a speedup of up to 2649× over VPR run with the fast option, at a cost of 36% average increase in minimum channel width for successful routing. However, the key limitation of this work is requiring an FPGA that is at least 400 times larger than the circuit being placed, making it unusable for large designs.

2. AI acceleration
AI technologies has been emerging in the past years to guide the annealing placer in choosing which type of move to make and greatly reduce computation time.

a. In terms of algorithmic logic (Software)
Reinforcement learning (RL)—is a branch of machine learning, it utilizes a software agent to make observations and takes actions within an environment, and its objective is to learn to act in a way that will maximize its expected long-term rewards. Consider the task of FPGA annealing placement [32, 52], the goal is to teach the annealer (agent) to make move decisions with reinforcement learning, given the current placement status (environment) S_t, RL techniques iteratively use an action value function $Q(s, a)$ to predict the immediate and future expected cost optimization(reward) if action a is chosen while the environment is in state s. Any action chosen will not only affect the immediate reward but also all the upcoming rewards and states. After performing action a_t and receiving reward r_{t+1}, the action value function Q can be updated as (Eq. 10.9):

$$Q(a_{t+1}) = Q(a_t) + \lambda(r_{t+1} - Q(a_t)) \tag{10.9}$$

where $r_{t+1} - Q(a_t)$ represents the deviation between the agent's estimate and the actual reward, and λ is the step size parameter to reduce this deviation.

b. In terms of computation architecture (Hardware)
   Scalar architecture (CPU)—Known studies deploy the agent on CPUs, such as [32, 52].
   In fact, competitive rivals such as vector (GPU) or matrix (TPU) engines could be even more efficient for this type of workload. However, none of these works is published by the time this book is written.

10.2.3 Analytical Placement Algorithms

Analytical placement methods consider global connectivity rather than evaluate the local modifications, however, the global minimum is usually an illegal placement with overlapping blocks, so constraints and heuristics must be applied to guide the algorithm to a legal solution. At the 2016 International Symposium on Physical Design (ISPD) contest, analytical placers occupied the top three positions (UTPlace [53], Ripple [54] and GPlace [55]).

Most modern FPGA analytical placers consist of the following three major actions:

Action 1: Global placement (GP). This action ignores some constraints (e.g., unit overlaps) and computes the best position (coordination) for each unit according to desired objectives (e.g., wirelength). It has a crucial impact on the overall placement quality.

Action 2: Legalization (LG). This action removes all overlaps among design units, assigning each of them into device units.

Action 3: Detailed placement (DP). This action further improves the legalized placement solution, typically in an iterative manner by rearranging a small number of units in a given region while keeping all other units fixed.

Among these actions, GP is the most time-consuming one. As a reference, elfPlace [56] reports that in terms of runtime breakdown, GP, LG, and DP consumes 59.8%, 19.9%, and 18.6%, respectively.

The criteria of analytical placement for FPGA application design includes:

1. Placement Schedule (Analytical)—Defines how to explore the solution space, or the algorithmic strategies of invoking each action.
2. Cost Function—It is used to evaluate the impact of each proposed status change based on the desired optimization objectives.
3. Computation Time—Having the same concerns with annealing techniques in the previous section, reducing placement time is an endless optimization direction for every researcher.

Placement Schedule (Analytical)

There are several different algorithmic concerns with FPGA analytical placement.

For GP, analytical solving process tends to pull FPGA units together for better PPA, so that the placement strategy is highly depends on the cost function, which will be discussed later in this section.

For LG, it is assumed that different PPA objectives are already optimized in the GP. Consequently, to preserve the GP result as far as possible, the objective of LG is to minimize the movement of the FPGA units. Partition-based methods are widely used due to their simplicity [54, 57, 58], during spreading, the FPGA will be evenly divided into a grid of x*y bins, and the legalizer will find an overflowed bin, expand it into a corresponding larger window, recursively partition the window and spread the units within it.

For DP, the objective is to find a better position for each FPGA unit in the available free spaces. These detailed refinements are often performed using low temperature simulated annealing to fully optimize the FPGA design [54, 59].

Cost Function

1. Wirelength
   Wirelength cost functions have two types of models to approximate HPWL: quadratic and nonlinear.
   Quadratic cost functions are mostly used for analytical placement [54, 57, 59], and the wirelength is modeled in quadratic formula:

$$\text{WireCost}_{\text{quadratic}} = \frac{1}{2}\sum_{i,j} w_{i,j}[(x_i - x_j)^2 + (y_i - y_j)^2] \tag{10.10}$$

   where $w_{i,j}$ is the weight of the connection between FPGA unit i and j. x and y denote the unit locations. The cost function can be written in matrix notation as:

$$\text{WireCost}_{\text{quadratic}} = \frac{1}{2}x^T Q_x x + c_x^T x + \frac{1}{2}y^T Q_y y + c_y^T y + \text{Constant} \tag{10.11}$$

   For the x dimension, a matrix, Q_x, represents connections between movable units (i.e., units being placed), and a vector, c_x, represents connections between movable and fixed units.
   The cost function can be further separated into x and y components and it is equivalent to solving:

$$Q_x + c_x = 0;\ Q_y + c_y = 0 \tag{10.12}$$

   Once formula (Eq. 10.12) is solved, the x and y locations of the FPGA units is settled.
   Nonlinear cost functions, on the other hand, using higher-order formula to represent wirelength. They formulate the nonoverlap constraint using differentiable nonlinear functions and solve it together with the wirelength in a unified objective function. Using nonlinear cost functions can often get higher solution quality, however, this quality improvement also comes with longer runtime due to the more expensive nonlinear optimization. Logarithm-sum-exponent (LSE) is a typical nonlinear expression to approximate the HPWL (Eq. 10.13).

$$\text{WireCost}_{\text{lse}} = \gamma(\log \sum_{v_i \in e} e^{\frac{x_i}{\gamma}} + \log \sum_{v_i \in e} e^{\frac{-x_i}{\gamma}}) \tag{10.13}$$

where e is the target net and $v_i \in e$ is the units in that net. When γ is equal to zero, the LSE model is reduced to the exact HPWL. However, in practical implementation, a small reasonable γ value is chosen to avoid arithmetic overflow.

2. Timing
   See timing cost of annealing methods.
3. Power
   See power cost of annealing methods.

Just like simulated annealing methods, total cost function will trades off these objectives above.

Computation Time

1. Parallel acceleration
   Since detailed placement can be seen as a low temperature annealing process, which we have discussed in the previous section, global placement and legalization can also benefit from parallel acceleration techniques:

   a. In terms of algorithmic logic (Software)
      Analytical solver for the x and y dimensions can be assigned to different computation unit. This parallelization could result in close to 2× improvement in computation time (with 2 CPU threads) [57].
   b. In terms of computation architecture (Hardware)
      Scalar architecture (CPU)—Using multiple CPU cores or threads as computation engines for analytical placement is the most ordinary way [57, 60].
      Vector architecture (GPU)—Nonlinear placement (using nonlinear cost function) usually performs better than quadratic placement (using quadratic cost function), although the difference in quality is small [61]. Acceleration of nonlinear analytical placement on GPUs then emerged [62, 63].
      Spatial architecture (FPGA)—*In the first work on acceleration of analytical placement on FPGAs [64], wirelength gradient is computed by using OpenCL.

2. AI acceleration
   AI technologies has been emerging in the past years to aid FPGA analytical placers to reduce computation time.

   a. In terms of algorithmic logic (Software)
      Deep learning (DL)—is a branch of machine learning, it uses data to train a model to make predictions from new data. For FPGA analytical placement, the first attempt [65] uses deep-learning to direct the placer's optimization strategy. In this work, a Convolutional Encoder-Decoder (CED) is utilized to predict the congestion present in subsequent placement iterations including the final placement, and achieve reductions in placer runtime between 27 and 40% with no significant deterioration in quality-of-result. [66] presents a CNN-inspired analytical placement algorithm to effectively handle the redundant frequency

translation problem for large-scale FPGAs.
Reinforcement learning (RL)—Discussed in previous section, RL framework can be integrated into detailed placement. In [67], several RL models are developed to capture the different characteristics of placement solutions and use them to guide decision making process. As a result, 50% of the CPU time is saved on the basis of [59].

b. In terms of computation architecture (Hardware)
Scalar architecture (CPU)—CPUs are still the most traditional computation platform for AI-aided FPGA analytical placement [65].
Works on vector (GPU) or matrix (TPU) computing platforms still needs time to emerge.

10.2.4 Summary and Trends

As the two major approaches for FPGA placement, annealing, and analytical engines have their own strengths and weaknesses.

1. For annealing
It has better quality for small designs, and easier to consider multiple objectives simultaneously, however, it is slower for large circuits, and sometimes could meet freezing problem—unable to escape local minima.
2. For analytical
The multi-stage (GP-LG-DP) characteristics make it a de facto hybrid placement technique that is highly flexible. It is more efficient and scalable for large designs with considerably good quality, however it still face its own challenges, such as targeting mix-size FPGA units, integrating timing/power metrics into the optimization objective. [68].
FPGA placement definitely will face new challenges continuously as FPGA architecture evolves. First, multiple objectives need to be considered at the same time due to the increasing complexity of modern FPGAs. It's usually difficult to make a trade-off among multiple objectives. Thus, an optimal resolution that performs good in every aspect will become harder to achieve. Secondly, extra architectural constraints are often extremely difficult to handle in placement due to their discreteness and irregularity. Last but not least, how to save computation time and memory footprint, especially when the FPGA design is gargantuan, still haunted every practitioners in this field.

10.3 Routing

10.3.1 Overview

FPGA routing is a process that determines which programmable switches should be turned on to connect all the logic unit input and output pins required by the circuit (Fig. 10.17).

Logic units in the application design must be well connected by using device's interconnection resources to actually make the circuit work, because a poor routing engine will lead to a lower maximum operating speed, greater power consumption, slower implementation time or even a failure to route all signals. The constraints is relatively simple: two nets cannot be routed on the same wire, which may cause routing congestion.

FPGA is like a city on a grid, routing is like solving the traffic problems in the city. Two fundamental steps must be carried out—building the traffic infrastructure (building routing resource graph under the help of routing guidance model) and navigating each trip (route all signals with routing engines).

Problem Definition: For each application design net, source is at which all nets begin, sink is at which all net terminals end. There can be one source (s_i) and multiple sinks ($T\_i = t\_i1, t12, \ldots t\_ik$, which refers to all k sinks of s_i), so the application design has set of sources ($S = s\_1, s\_2, \ldots s\_i$) and set of sets of sinks ($T = T\_1, T\_2, \ldots T\_i, T\_i = t\_i1, t12, \ldots t\_ik$). Routing problem is to find paths from each source s_i to all sinks in T_i, paths emanating from different sinks that must be disjoint (cannot share any nodes or edges).

Similar to placement, routing is extremely important for FPGA application design, since a poor solution will lead to a lower maximum operating speed, increase power consumption, slow implementation time or even a failure to route all signals. Given the fact that routing is a NP complete problem, finding a good routing is challenging, let alone relative scarcity of routing resources and signals will compete for the same routing resources.

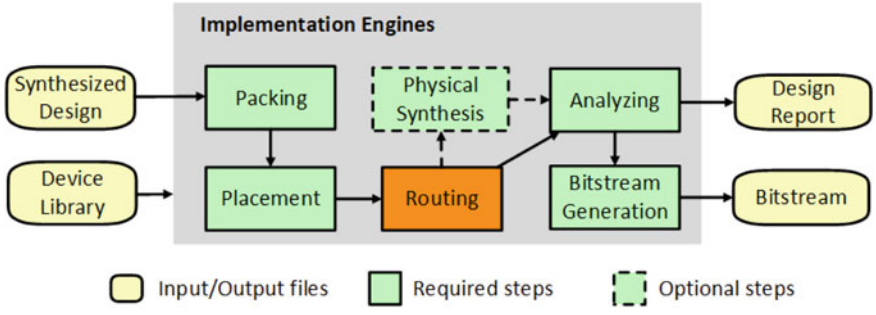


Fig. 10.17 Routing's position in FPGA application EDA flow

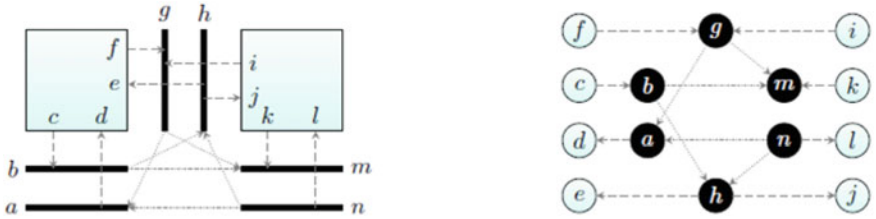


Fig. 10.18 FPGA RRG example [69]

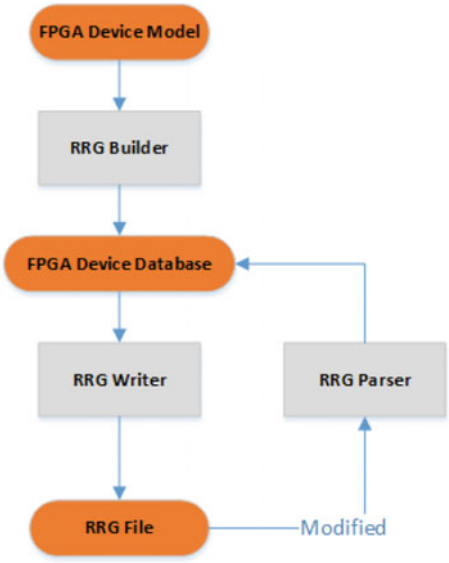


Fig. 10.19 FPGA RRG processing flow

Routing Resource Graph: The primary data structure representing FPGA routing resources is the directed Routing Resource Graph (RRG) $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex $v \in V$ represents wires and pins, each edge $e_{ij} \in E$ represents a programmable connection between a pin and a wire segment, or a programmable routing switch between two wire segments (Fig. 10.18). Programmable switches can be fabricated as pass transistors, tri-state buffers, or multiplexers. Multiplexers are the dominant form of programmable interconnect in recent FPGAs due to a superior area-delay product and thus unidirectional.

RRG can be represented by readable file. VTR-XML [70] is the only publicly known format to describe RRG. The RRG building, parsing and writing flow can be described as (Fig. 10.19).

Mainstream Algorithms: Routing engines can be categorized from different perspectives:

1. By algorithmic logic
   Negotiation-based routing algorithms[69, 71] typically consist of two major steps: path searching and congestion removal. During path searching, it explore all feasible paths for each net and identify the best one. After path searching, rip-up and rerouting operation is carried out to eliminate congestion. Boolean-based routing algorithms [72–76] transform the FPGA routing task to one of simultaneously satisfying a set of Boolean constraints. If successful, the solution found by the solver is converted into a valid route, otherwise the signal is unroutable.
   The negotiation-based routing algorithm is in dominant use in the FPGA community due to its superior performance and quality of results.
2. By optimization objectives
   Routability-driven routing prioritizes congestion avoidance as its primary goal[77]; timing-driven routing maximizes the circuit speed[69, 78, 79]; power-aware routing saves the power consumption [43, 80], etc. An excellent routing engine would always balance well among these objectives.
3. By computation acceleration
   In order to reduce compilation time while maintaining quality of results, routing computations can be accelerated either by parallelism or by AI.
4. By targeted architecture
   For other special modern FPGA architecture features (such as advanced package (2.5D/3D), complexed clocking), there are additional routing considerations to adapt to them.

10.3.2 Negotiation-Based Routing Algorithms

Negotiation-based routing methods route nets one by one. Larry McMurchie and Carl Ebling proposed a negotiation approach in 1995, called Pathfinder [71], which is very famous in routing society. Pathfinder first gives every edge (connection) in RRG a cost that depends on current usage and historical usage, then each net is routed by a breath first searching (BFS), making the cost the lowest. Multiple nets may use the same node in RRG (flagged as a congestion). A congestion node is given a higher cost. If a route must include a congested node, it will "negotiate" with the other routes and make them go around (rip-up and re-route).

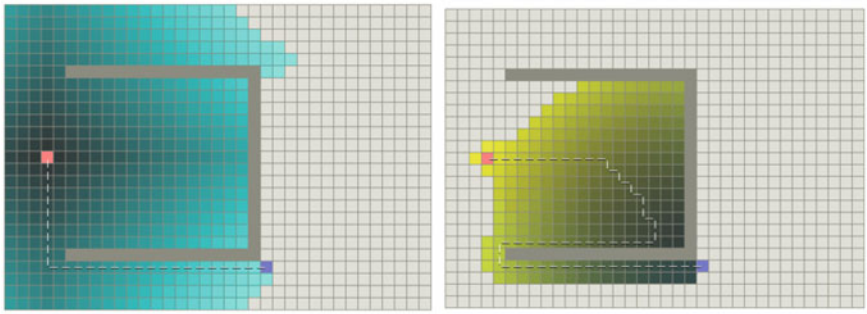
The criteria of negotiation-based routing for FPGA application design includes:

1. Routing Schedule (Negotiation-based)—Defines how to explore the solution space, or the algorithmic strategies of invoking each action.
2. Cost Function—It is used to evaluate the impact of each proposed status change based on the desired optimization objectives.

(a) Search the path from the source to the first sink

(b) Set the cost of nodes on the previous path to 0 and search for the second sink

Fig. 10.20 Routing schedule for multiple sinks in the same net



(a) BFS without direction

(b) BFS with direction

Fig. 10.21 When BFS meets a concave obstacle

3. Computation Time—Having the same concerns with annealing techniques in the previous section, reducing placement time is an endless optimization direction for every researcher.

Routing Schedule (Negotiation-based)

Route nets in decreasing order of fanout is a common schedule, because high-fanout nets tend to span the larger area of FPGA and easier to route when there is less congestion to other nets routed earlier, low fanout nets tend to be more localized and relatively easier to route even in the presence of some congestion.

When routing a net with one source and multiple sinks, breath first searching (BFS) is invoked to find the first sink like the process of wave expansion. The wavefront always have the biggest cost. After that, the cost of nodes on the previous path is set to 0 and the second sink is searched iteratively (Fig. 10.20).

In general, BFS without direction is time-consuming and directed search is far more effective, however, when the obstacle is concave, BFS without direction finds the best route but time-consuming, BFS with direction is time saving but sometimes can not find the best route (Fig. 10.21).

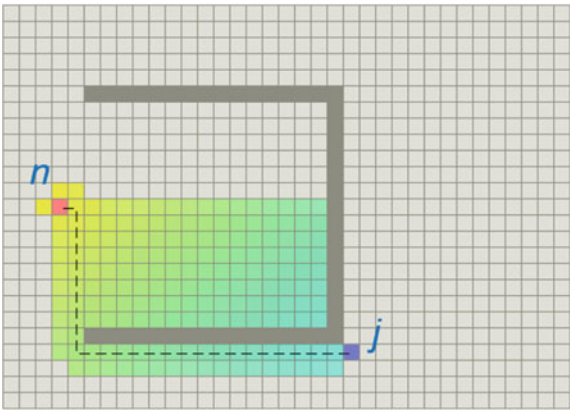


Fig. 10.22 Banlanced BFS searching in FPGA routing

Therefore, cost function must be proper balanced between directed searching and directionless searching. Just like authoritarian and democracy, a good political governance model would always balance them well (Fig. 10.22).

The general negotiation process's pseudo code is shown in Algorithm 6.

```
Let: RT_i be the set of nodes in the current routing of net i;
while shared resources exist(routing congestion occurs) do
    foreach net, i do
        rip-up routing tree RT_i;
        RT_i = s_i;
        foreach sinkt_ij do
            Initialize priority queue PQ to RT_i at cost 0;
            while sink t_ij not found do
                Remove lowest cost node m from PQ;
                foreach fan-out node n of node m do
                    Add n to PQ at PathCost(n) = Costn + PathCost(m);
                end
            end
            foreach node n in path t_ij to s_i do
                Update c_n;
                Add n to RT_i;
            end
        end
    end
    update historical cost for n;
end
```

Algorithm 6: Negotiation process in negotiation-based routing algorithm (pseudo-code)

In VTR routing, this consideration is expressed by adding an ExpectedCost(n, j) to the PathCost(n) (Eq. 10.14).

$$\text{TotalCost}(n) = \text{PathCost}(n) + \alpha \cdot \text{ExpectedCost}(n, j) \tag{10.14}$$

where ExpectedCost(n, j) is an estimated cost from current node n to target sink j, PathCost(n) is the cost of the path from the current partial routing tree to node n.

Cost Function

1. Routability (Congestion)
   In real life, there has been a system of surcharging for use of public roads that are subject to congestion through excess demand. This traffic control measure has been applied to many big cities (Fig. 10.23).
   Pathfinder [71] has set a classic example of routability cost, for node n in the RRG, the cost function is presented as (Eq. 10.15).

$$\text{RoutabilityCost}(n) = (b(n) + h(n)) \cdot p(n) \tag{10.15}$$

   where $b(n)$ is the base cost of a routing through node n, $h(n)$ is related to the history of congestion on node n during previous iterations, and $p(n)$ is related to the number of nets (signals) presently routed through node n at the current iteration.
   VTR router [69] is based upon the Pathfinder negotiated congestion algorithm, the modified routability cost it defined is shown as (Eq. 10.16).

$$\text{RoutabilityCost}(n) = b(n) \cdot h(n) \cdot p(n) + \text{Bend}(n, m) \tag{10.16}$$

   where $b(n), h(n)$ and $p(n)$ has exactly the same meaning with those in the Pathfinder, the additional parameter Bend (n, m) penalize global routes with bends since these routes are less likely to use long wires, making detailed routes difficult to implement (congestion more likely to occur). Meanwhile, multiplying $b(n)$ and $h(n)$ eliminates normalization.
2. Timing
   Cost of using a RRG node n in the routing of a connection c can be represented as (Eq. 10.17).

$$\text{TimingCost}(n) = \text{TimingCriticality}(c) \cdot \text{Delay}(n) \tag{10.17}$$

   The timing criticality is the ratio of the connection slack to the longest delay in the circuit (Eq. 10.18).

$$\text{TimingCriticality}(c) = 1 - \frac{\text{Slack}(c)}{\text{Delay}_{\max}} \tag{10.18}$$

   where $\text{Delay}_{\max}$ is the critical path delay (maximum arrival time of all sinks in the circuit), and Slack(c) is the amount of delay that can be added to the connection c without increasing the critical path delay.
   In Pathfinder, it uses TimingCriticality(c) as the weighted factor of timing cost.

3. Power
   Lamoureaux [43] modified VPR's cost function by adding a power cost (Eq. 10.19):

$$\text{PowerCost}(n) = \text{ActCriticality}(i) \cdot \text{Cap}(n) \tag{10.19}$$

   where $\text{Cap}(n)$ is the capacitance associated with routing resource node n and $\text{ActCriticality}(i)$ is the activity criticality of net i.
   In [43], it uses $\text{ActCriticality}(i)$ as the weighted factor of timing cost.

$$\text{ActCriticality}(i) = \min\left(\frac{\text{Act}(i)}{\text{MaxAct}}, \text{MaxActCriticality}\right) \tag{10.20}$$

   where $\text{Act}(i)$ is the switching activity in net i, MaxAct is the maximum switching activity of all the nets, and MaxActCriticality is the maximum activity criticality that any net can have.

Total cost function often trades off these objectives above. If the trade-off variable λ determines how much weight to give timing cost, variable γ determines how much weight to give power cost, then the total cost could be described as (Eq. 10.21).

$$\text{WeightedCost} = \lambda\text{TimingCost} + \gamma\text{PowerCost} + (1 - \lambda - \gamma)\text{RoutabilityCost} \tag{10.21}$$

Computation Time

1. Parallel acceleration
   Researchers have been tirelessly looking for ways to accelerate FPGA routing through parallelism, since routing is one of the most time-consuming compilation step in the whole flow. Most of the parallel acceleration work for FPGA routing focuses on the negotiation-based methods. An ideal parallel router is not only fast, but also scalable and deterministic.

Fig. 10.23 Congestion charging in central London

a. In terms of algorithmic logic (Software)
Almost all parallel routers are either coarse-grain or fine-grain. Coarse-grain parallel routers [81–83] distribute the problem by partitioning the nets and then route them independently, whilst fine-grain parallel routers accelerate the routing of a single net. For coarse-grain parallel routers, nets are partitioned based on the independence of their bounding boxes. If the bounding boxes of two nets overlap, the possibility of conflicts between the nets is high and they should probably not be routed in parallel. For fine-grain parallel routers [84], works for an individual net such as maze expansion could be accelerated.

b. In terms of computation architecture (Hardware)
Scalar architecture (CPU)—Most of the parallelization work on FPGA routing is based on CPU [81–83].
Vector architecture (GPU)—GPU-accelerated EDA has been studied for years [85], however, the first published work on utilizing GPU to accelerate FPGA routing [86] came out later in 2017. It leverage Bellman-Ford algorithm to optimize the computation and experimental results show that an average of 18.72% speedup is achieved.
Spatial architecture (FPGA)—[87] is the first attempt on FPGA-accelerated FPGA routing, however, it ends up with 4–6× slower than running on pure CPU platform due to the limited performance of the chosen hardware platform (mid-end ARM+FPGA SoC).

2. AI acceleration
FPGA negotiation-based router start to benefit from AI technologies from the 2010s.

a. In terms of algorithmic logic (Software)
Reinforcement learning (RL)—The agent is guided toward achieving legal routing solution by formulating a reward function. Every time an action is taken by an agent while moving from one node to another, it must get a reward that is added to its experience for any future moves regarding that particular node. In [88], the reward formulation is defined as (Eq. 10.22), showing that the objective of the routing is to minimize the number of conflicts.

$$r_t = -\Delta_{\text{conflict}} \tag{10.22}$$

The agent continuously learns and adjusts the award values until get the optimized solution. It is reported that on average, the RL technique can reduce 30% routing time for similar quality of results.

b. In terms of computation architecture (Hardware)
Scalar architecture (CPU)—CPUs are still the most traditional computation platform for AI-aided FPGA negotiation-based routing.
Works on vector (GPU) or matrix (TPU) computing platforms still needs time to emerge.

10.3.3 Summary and Trends

Modern FPGA routing are essentially RRG searching problem, balancing concerned metrics such as routability, timing, and power. Negotiation-based algorithms have been proved to be the most efficient solvers, and hence have ruled the FPGA routing world for years. VTR is the most popular open-sourced implementation of negotiation-based FPGA router. How to further refine it in terms of QoR or computation time has been the main researching topic and this trend will continue in the foreseeable future.

AI(ML)-aided techniques has emerged in accelerating FPGA routing tasks. These intelligent routing algorithms can significantly improve the efficiency and reliability of FPGA designs, while also reducing the design time and cost [89]. As the study deepens, AI-aided router will play an increasingly important role.

References

1. V. Betz, J. Rose, VPR: a new packing, placement and routing tool for FPGA research. FPL (1997)
2. A.S. Marquardt, V. Betz, J. Rose, Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density, in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '99. (Association for Computing Machinery, New York, NY, USA, 1999), pp. 37–46. [Online]. Available: https://doi.org/10.1145/296399.296426
3. E. Bozorgzadeh, S. Ogrenci-Memik, M. Sarrafzadeh, Rpack: Routability-driven packing for cluster-based FPGAs, in *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)* (2001), pp. 629–634
4. A. Singh M. Marek-Sadowska, 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*, ser. FPGA '02. (Association for Computing Machinery, New York, NY, USA, 2002), pp. 59–66. [Online]. Available: https://doi.org/10.1145/503048.503058
5. J. Luu, J. H. Anderson, J.S. Rose, 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*, ser. FPGA '11. (Association for Computing Machinery, New York, NY, USA, 2011), pp. 227–236. [Online]. Available: https://doi.org/10.1145/1950413.1950457
6. J. Luu, J. Rose, J. Anderson, Towards interconnect-adaptive packing for FPGA, in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '14 (Association for Computing Machinery, New York, NY, USA, 2014), pp. 21–30. [Online]. Available: https://doi.org/10.1145/2554688.2554783
7. W. Feng, K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint, in *2012 International Conference on Field-Programmable Technology* (2012), pp. 8–15
8. W. Feng, J. Greene, K. Vorwerk, V. Pevzner, A. Kundu, Rent's rule based FPGA packing for routability optimization, in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '14 (Association for Computing Machinery, New York, NY, USA, 2014), pp. 31–34. [Online]. Available: https://doi.org/10.1145/2554688.2554763

9. D.T. Chen, K. Vorwerk, A. Kennings, Improving timing-driven FPGA packing with physical information, in *2007 International Conference on Field Programmable Logic and Applications* (2007), pp. 117–123.
10. Z. Huang, Z. Li, N. Wang, P. Tao, X. Zhou, L. Wang, Repack: a packing algorithm to enhance timing and routability of a circuit, in *2012 IEEE 11th International Conference on Solid-State and Integrated Circuit Technology* (2012), pp. 1–5
11. S.T. Rajavel, A. Akoglu, Mo-pack: many-objective clustering for FPGA CAD, in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2011), pp. 818–823
12. K.E. Murray, O. Petelin, S. Zhong, J.M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A.G. Graham, J. Wu, M.J.P. Walker, H. Zeng, P. Patros, J. Luu, K.B. Kent, V. Betz, Vtr 8: high-performance cad and customizable FPGA architecture modelling. ACM Trans. Reconfigurable Technol. Syst. **13**(2) (2020). [Online]. Available: https://doi.org/10.1145/3388617
13. G. Karypis, V. Kumar, Multilevel k-way hypergraph partitioning, in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, ser. DAC '99 (Association for Computing Machinery, New York, NY, USA, 1999), pp. 343–348. [Online]. Available: https://doi.org/10.1145/309847.309954
14. J. Luu, Architecture-aware packing and cad infrastructure for field-programmable gate arrays, Ph.D. dissertation, University of Toronto (2014)
15. K. Keutzer, Dagon: technology binding and local optimization by DAG matching, in *24th ACM/IEEE Design Automation Conference* (1987), pp. 341–347
16. G. Chen, J. Cong, Simultaneous timing driven clustering and placement for FPGAS, in *International Conference on Field Programmable Logic and Applications* (Springer, 2004), pp. 158–167
17. Z. Marrakchi, H. Mrabet, H. Mehrez, Hierarchical FPGA clustering based on multilevel partitioning approach to improve routability and reduce power dissipation, in *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)* (2005), pp. 4–25
18. G. Karypis, V. Kumar, Hmetis: a hypergraph partitioning package. ACM Trans. Architect. Code Optim. (1998)
19. H. Zhang, L. Song, Z. Han, Y. Zhang, Basics of hypergraph theory, in *Hypergraph Theory in Wireless Communication Networks* (Springer, 2018), pp. 1–19
20. S. Yang, Logic synthesis and optimization benchmarks user guide: version 3.0. Citeseer (1991)
21. S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, P. Sanders, High-quality hypergraph partitioning. ACM J. Exp. Algorithmics (2022). [Online]. Available: https://doi.org/10.1145/3529090
22. X. Tang, P.-E. Gaillardon, G. De Micheli, Pattern-based FPGA logic block and clustering algorithm, in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), pp. 1–4
23. P.-E. Gaillardon, X. Tang, G. Kim, G. De Micheli, A novel FPGA architecture based on ultrafine grain reconfigurable logic cells. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **23**(10), 2187–2197 (2015)
24. P. Maidee, C. Ababei, K. Bazargan, Fast timing-driven partitioning-based placement for island style FPGAs, in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)* (2003), pp. 598–603
25. A. Khatkhate, C. Li, A.R. Agnihotri, M.C. Yildiz, S. Ono, C.-K. Koh, P.H. Madden, Recursive bisection based mixed block placement, in *Proceedings of the 2004 International Symposium on Physical Design*, ser. ISPD '04 (Association for Computing Machinery, New York, NY, USA, 2004), pp. 84–89. [Online]. Available: https://doi.org/10.1145/981066.981084
26. J. Zhao, Q. Zhou, Y. Cai, Fast congestion-aware timing-driven placement for island FPGA, in *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits and Systems* (2009), pp. 24–27
27. Intel, Intel Quartus Prime pro edition user guide: design compilation. https://www.intel.com/content/www/us/en/programmable/documentation/zpr1513988353912.html.
28. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines **3** (1953)

29. J. Yuan, J. Chen, L. Wang, X. Zhou, Y. Xia, J. Hu, Arbsa: adaptive range-based simulated annealing for FPGA placement. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **38**(12), 2330–2342 (2019)
30. K.E. Murray, O. Petelin, S. Zhong, J.M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A.G. Graham, J. Wu, M.J.P. Walker, H. Zeng, P. Patros, J. Luu, K.B. Kent, V. Betz, Vtr 8: high-performance cad and customizable FPGA architecture modelling. ACM Trans. Reconfigurable Technol. Syst. **13**(2) (2020). [Online]. Available: https://doi.org/10.1145/3388617
31. K. Vorwerk, A. Kennings, J.W. Greene, Improving simulated annealing-based FPGA placement with directed moves. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **28**(2), 179–192 (2009)
32. M.A. Elgammal, K.E. Murray, V. Betz, Rlplace: using reinforcement learning and smart perturbations to optimize FPGA placement. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **41**(8), 2532–2545 (2022)
33. A. Ludwin, V. Betz, Efficient and deterministic parallel placement for FPGAs **16**(3) (2011). [Online]. Available: https://doi.org/10.1145/1970353.1970355
34. T. Kong, A novel net weighting algorithm for timing-driven placement, in *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002* (2002), pp. 172–176
35. H. Ren, D. Pan, D. Kung, Sensitivity guided net weighting for placement-driven synthesis. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **24**(5), 711–721 (2005)
36. A. Marquardt, V. Betz, J. Rose, Timing-driven placement for FPGAs in *FPGA '00* (2000)
37. K. Eguro, S. Hauck, Enhancing timing-driven FPGA placement for pipelined netlists, in *2008 45th ACM/IEEE Design Automation Conference* (2008), pp. 34–37
38. C. Guth, V. Livramento, R. Netto, R. Fonseca, J.L. Güntzel, L. Santos, Timing-driven placement based on dynamic net-weighting for efficient slack histogram compression, in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15 (Association for Computing Machinery, New York, NY, USA, 2015), pp. 141–148. [Online]. Available: https://doi.org/10.1145/2717764.2717766
39. W. Swartz, C. Sechen, Timing driven placement for large standard cell circuits, in *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, ser. DAC '95 (Association for Computing Machinery, New York, NY, USA, 1995), pp. 211–215. [Online]. Available: https://doi.org/10.1145/217474.217531
40. A. Chowdhary, K. Rajagopal, S. Venkatesan, T. Cao, V. Tiourin, Y. Parasuram, B. Halpin, How accurately can we model timing in a placement engine? in *Proceedings 42nd Design Automation Conference* (2005) pp. 801–806
41. T. Luo, D. Newmark, D.Z. Pan, A new lp based incremental timing driven placement for high performance designs, in *2006 43rd ACM/IEEE Design Automation Conference* (2006), pp. 1115–1120
42. N. Viswanathan, G.-J. Nam, J. A. Roy, Z. Li, C. J. Alpert, S. Ramji, C. Chu, Itop: integrating timing optimization within placement, in *Proceedings of the 19th International Symposium on Physical Design*, ser. ISPD '10 (Association for Computing Machinery, New York, NY, USA, 2010), pp. 83–90. [Online]. Available: https://doi.org/10.1145/1735023.1735048
43. J. Lamoureux, S.J.E. Wilton, On the interaction between power-aware FPGA cad algorithms, in *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)* (2003) pp. 701–708
44. W.-J. Sun, C. Sechen, A parallel standard cell placement algorithm. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **16**(11), 1342–1357 (1997)
45. S. Birk, J.G. Steffan, J.H. Anderson, Parallelizing FPGA placement using transactional memory, in *2010 International Conference on Field-Programmable Technology* (2010), pp. 61–69
46. J.B. Goeders, G.G. Lemieux, S.J. Wilton, Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition, in *2011 International Conference on Reconfigurable Computing and FPGAs* (2011), pp. 41–48
47. B. Huang, H. Zhang, Application of multi-core parallel computing in FPGA placement, in *2013 2nd International Symposium on Instrumentation and Measurement, Sensor Network and Automation (IMSNA)* (2013), pp. 884–889

48. M. An, J.G. Steffan, V. Betz, Speeding up FPGA placement: parallel algorithms and methods, in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines* (2014), pp. 178–185
49. C. Fobel, G. Grewal, D. Stacey, A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and FPU architectures, in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), pp. 1–8
50. A. Al-Kawam, H.M. Harmanani, A parallel GPU implementation of the timber wolf placement algorithm, in *2015 12th International Conference on Information Technology - New Generations* (2015), pp. 792–795
51. M.G. Wrighton, A.M. DeHon, Hardware-assisted simulated annealing with application for fast FPGA placement, in *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '03 (Association for Computing Machinery, New York, NY, USA, 2003), pp. 33–42. [Online]. Available: https://doi.org/10.1145/611817.611824
52. C. Tian, L. Chen, Y. Wang, S. Wang, J. Zhou, Y. Zhang, G. Li, Improving simulated annealing algorithm for FPGA placement based on reinforcement learning, in *2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, vol. 10 (2022), pp. 1912–1919
53. W. Li, S. Dhar, D.Z. Pan, Utplacef: a routability-driven FPGA placer with physical and congestion aware packing. IEEE Trans. Comput-Aid. Des. Integr. Circ. Syst. **37**(4), 869–882 (2018)
54. C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E.F.Y. Young, B. Yu, RippleFPGA: a routability-driven placement for large-scale heterogeneous FPGAs, in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2016), pp. 1–8
55. R. Pattison, Z. Abuowaimer, S. Areibi, G. Gráwal, A. Vannelli, Gplace: a congestion-aware placement tool for ultrascale FPGAs, in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2016), pp. 1–7
56. Y. Meng, W. Li, Y. Lin, D.Z. Pan, "elf place: electrostatics-based placement for large-scale heterogeneous FPGAs. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **41**(1), 155–168 (2022)
57. M. Gort, J.H. Anderson, Analytical placement for heterogeneous FPGAs, in *22nd International Conference on Field Programmable Logic and Applications (FPL)* (2012) pp. 143–150
58. D. Vercruyce, E. Vansteenkiste, D. Stroobandt, Liquid: high quality scalable placement for large heterogeneous FPGAs, in *2017 International Conference on Field Programmable Technology (ICFPT)* (2017), pp. 17–24
59. Z. Abuowaimer, D. Maarouf, T. Martin, J. Foxcroft, G. Gréwal, S. Areibi, A. Vannelli, Gplace3.0: routability-driven analytic placer for ultrascale FPGA architectures. ACM Trans. Des. Autom. Electron. Syst. **23**(5) (2018). [Online]. Available: https://doi.org/10.1145/3233244
60. W. Li, D.Z. Pan, A new paradigm for FPGA placement without explicit packing. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **38**(11), 2113–2126 (2019)
61. T. Lin, C. Chu, Polar 2.0: an effective routability-driven placer, in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)* (2014), pp. 1–6
62. R. Pattison, C. Fobel, G. Grewal, S. Areibi, Scalable analytic placement for FPGA on GPGPU, in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (2015), pp. 1–6
63. C.-X. Lin, M.D.F. Wong, Accelerate analytical placement with GPU: a generic approach, in *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)* (2018), pp. 1345–1350
64. S. Dhar, L. Singhal, M.A. Iyer, D.Z. Pan, FPGA-accelerated spreading for global placement, *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (2019), pp. 1–7
65. A. Al-Hyari, A. Shamli, T. Martin, S. Areibi, G. Grewal, An adaptive analytic FPGA placement framework based on deep-learning, in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)* (2020), pp. 3–8

66. H. Wang, X. Tong, C. Ma, R. Shi, J. Chen, K. Wang, J. Yu, Y.-W. Chang, CNN-inspired analytical global placement for large-scale heterogeneous FPGAs, in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22 (Association for Computing Machinery, New York, NY, USA, 2022), pp. 637–642
67. P. Esmaeili, T. Martin, S. Areibi, G. Grewal, Guiding FPGA detailed placement via reinforcement learning, in *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)* (2022), pp. 1–6
68. T. Liang, G. Chen, J. Zhao, S. Sinha, W. Zhang, AMF-placer 2.0: open source timing-driven analytical mixed-size placer for large-scale heterogeneous FPGA (2022)
69. K.E. Murray, S. Zhong, V. Betz, Air: A fast but lazy timing-driven FPGA router, in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2020), pp. 338–344
70. V. Developers, Routing resource graph-vtr. https://docs.verilogtorouting.org/en/latest/api/vpr/rr_graph/
71. L. McMurchie, C. Ebeling, Pathfinder: a negotiation-based performance-driven router for FPGAs, in *Third International ACM Symposium on Field-Programmable Gate Arrays* (1995), pp. 111–117
72. R. Wood, R. Rutenbar, FPGA routing and routability estimation via Boolean satisfiability. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **6**(2), 222–231 (1998)
73. G.-J. Nam, F. Aloul, K. Sakallah, R. Rutenbar, A comparative study of two Boolean formulations of FPGA detailed routing constraints. IEEE Trans. Comput. **53**(6), 688–696 (2004)
74. S. Mukherjee, S. Roy, Sat based multi pin net detailed routing for FPGA, in *2010 International Symposium on Electronic System Design* (2010), pp. 141–146
75. V. Chopra, C. Deptt, P. India, A. Singh, P.P. India, Ant colony optimization approach for solving FPGA routing with minimum channel width (2011)
76. H. Fraisse, A. Joshi, D. Gaitonde, A. Kaviani, Boolean satisfiability-based routing and its application to Xilinx ultrascale clock network, in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16 (Association for Computing Machinery, New York, NY, USA, 2016), pp. 74–79. [Online]. Available: https://doi.org/10.1145/2847263.2847342
77. S. Boshra, H. Abbas, A. Darwish, I. Talkhan, Performance and routability improvements for routability-driven FPGA routers, in *2006 IEEE International Symposium on Circuits and Systems (ISCAS)* (2006) pp. 4
78. D. Vercruyce, E. Vansteenkiste, D. Stroobandt, Croute: a fast high-quality timing-driven connection-based FPGA router, in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2019), pp. 53–60
79. Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, D. Stroobandt, RWRoute: an open-source timing-driven router for commercial FPGAs. ACM Trans. Reconfigurable Technol. Syst. **15**(1) (2021). [Online]. Available: https://doi.org/10.1145/3491236
80. C.H. Hoo, Y. Ha, A. Kumar, A directional coarse-grained power gated FPGA switch box and power gating aware routing algorithm, in *2013 23rd International Conference on Field programmable Logic and Applications* (2013), pp. 1–4
81. C. H. Hoo, A. Kumar, Y. Ha, Paralar: A parallel FPGA router based on Lagrangian relaxation, in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)* (2015), pp. 1–6.
82. D. Wang, Z. Duan, C. Tian, B. Huang, N. Zhang, Parra: a shared memory parallel FPGA router using hybrid partitioning approach. IEEE Trans Comput-Aided Des Integrated Circ Syst **39**(4), 830–842 (2020)
83. M. Shen, G. Luo, N. Xiao, Coarse-grained parallel routing with recursive partitioning for FPGAs. IEEE. Trans. Parallel Distrib. Syst. **32**(4), 884–899 (2021)
84. Y. Moctar, M. Stojilović, P. Brisk, Deterministic parallel routing for FPGAs based on Galois parallel execution model, in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018) pp. 21–214.
85. J.F. Croix, S.P. Khatri, Introduction to GPU programming for EDA, in *2009 IEEE/ACM International Conference on Computer-Aided Design—Digest of Technical Papers* (2009) pp. 276–280

86. M. Shen, G. Luo, Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17 (Association for Computing Machinery, New York, NY, USA, 2017) pp. 105–114. [Online]. Available: https://doi.org/10.1145/3020078.3021732
87. D. Korolija, M. Stojilović, FPGA-assisted deterministic routing for FPGAs, in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019), pp. 155–162
88. U. Farooq, N. Ul Hasan, I. Baig, M. Zghaibeh, Efficient FPGA routing using reinforcement learning, in *2021 12th International Conference on Information and Communication Systems (ICICS)* (2021), pp. 106–111
89. T. Martin, C. Barnes, G. Grewal, S. Areibi, Integrating machine-learning probes into the VTR FPGA design flow, in *2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)* (2022), pp. 1–6