CHAPTER 14

PLACEMENT FOR GENERAL-PURPOSE FPGAS

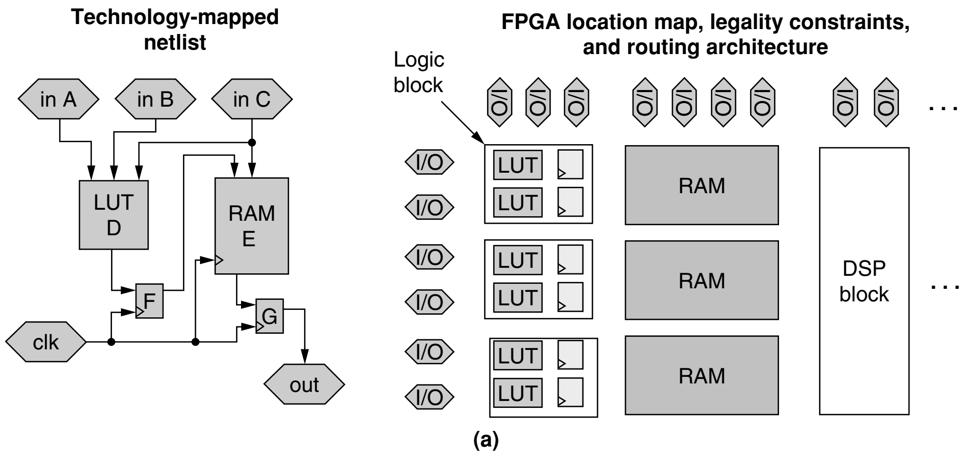
Vaughn Betz
Altera Corporation

Placement follows technology mapping in the CAD flow and chooses a location for each block in a circuit. This chapter describes "general-purpose" placement approaches; these techniques can be used with any circuit targeting the commercial field-programmable gate arrays (FPGAs) in widespread use today. After defining the placement problem and optimization goals, the chapter describes the clustering algorithms that are frequently used in conjunction with placement tools. Three different classes of placement algorithms are then detailed: simulated annealing, partition based, and analytic. The chapter concludes with suggestions for further reading and open challenges in FPGA placement.

14.1 THE FPGA PLACEMENT PROBLEM

An FPGA placement algorithm takes two basic inputs: (1) a netlist specifying the functional blocks to be implemented and the connections between them, and (2) a device map indicating which functional unit can be placed at each location. The algorithm selects a legal location for each block such that the circuit wiring is optimized. Figure 14.1 illustrates the FPGA placement problem. Both the legality constraints and the optimization metric (what constitutes a "good" arrangement of functional blocks) depend on the FPGA architecture being targeted.

A good placement is extremely important for FPGA designs—without a high-quality placement, a circuit generally cannot be successfully routed. Even if the circuit does route, a poor placement will still lead to a lower maximum operating speed and increased power consumption. At the same time, finding a good placement for a circuit is a challenging problem. A large commercial FPGA contains approximately 500,000 functional blocks, leading to approximately 500,000! possible placements. Exhaustive evaluation of the placement solution space is therefore impossible. Furthermore, placement is a computationally hard problem, so there are no known algorithms that produce optimal results in practical central processing unit (CPU) time. Consequently, the development of fast and effective heuristic placement algorithms is a very important research area.


Technology-mapped netlist
in A
in B
in C
LUT D
RAM E
F
G
clk
out
FPGA location map, legality constraints, and routing architecture
Logic block
I/O
LUT
RAM
DSP block
(a)

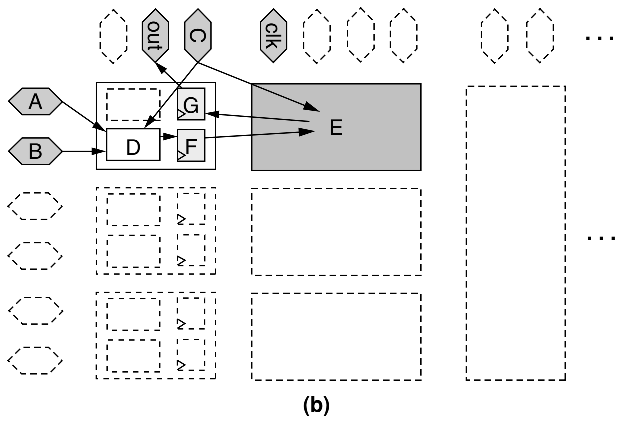


FIGURE 14.1 ■ Placement overview: (a) inputs to the placement algorithm, and (b) placement algorithm output—the location of each block.

14.1.1 Device Legality Constraints

The fact that all resources are prefabricated in an FPGA leads to a variety of placement legality constraints:

- A legal placement must place a functional block only in a location on the chip that can accommodate it. For example, a RAM block must be placed in a RAM location, and a lookup table (LUT) must be placed in a LUT location.
- Usually there are legality constraints on groups of functional blocks. In Altera's Stratix-II FPGAs, for example, a *logic block* contains 16 LUTs and 16 registers [1]. However, there are limits on the number of clock signals, clock enable signals, and routing inputs to the logic block. Consequently, not every grouping of 16 LUTs and 16 registers constitutes

a legal logic block, and the placement algorithm must ensure that it does not produce illegal logic blocks.

- Some groups of functional blocks must be placed in a specific relative orientation so that they can make use of special, dedicated routing resources. The simplest example of this constraint is arithmetic logic cells—in order to use the dedicated carry-chain hardware available in an FPGA, the logic cells forming a carry chain must be placed adjacent to each other in the sequence required by the carry structure.
- There are other detailed legality constraints, such as a limit on the number of global clocking resources in each area of the device, which commercial FPGA placement algorithms must respect.[1]

14.1.2 Optimization Goals

The basic goal of an FPGA placement algorithm is to locate functional blocks such that the interconnect required to route the signals between them is minimized. As Figure 14.2 illustrates, the routing required to connect two blocks is a function not only of the distance between them but also of the FPGA architecture. Figure 14.2(a) shows the wiring required to connect two blocks in different relative positions in a Stratix-II FPGA. Stratix-II is an *island-style* FPGA [3] that contains routing segments that span 4, 16, and 24 logic blocks. Programmable switches allow routing segments in the same direction (horizontal or vertical) to be connected at their endpoints to create longer routes. Other programmable switches allow some horizontal routing segments to connect to vertical routing segments where they cross and vice versa. In an island-style FPGA, the amount of wiring required to connect two functional blocks is roughly proportional to the Manhattan distance between them.

Figure 14.2(b) shows that the wiring required by the same placements in an FPGA with a *hierarchical* routing architecture (in this case the Altera APEX family [4]) is quite different. For hierarchical FPGAs, the amount of wiring required to connect two functional blocks is proportional to the number of levels of the routing hierarchy that must be traversed to connect them. Note that even the ranking of placement choices is different between APEX and Stratix-II—in Stratix-II placements, *A* and *C* are best, while in APEX placements, *A* and *B* are best. Clearly FPGA placement algorithms must have a model of the routing architecture they target in order to achieve good results.

FPGA placement tools can broadly be divided into *routability-driven* and *timing-driven* algorithms. Routability-driven algorithms try to create a placement that minimizes the total interconnect required, as this increases the probability of successfully routing the design. Since FPGA interconnect is prefabricated, the amount of interconnect in each region of a device is fixed, and a placement that requires more interconnect in a device region than that region contains cannot be routed. Consequently, some routability-driven placement algorithms

[1] Researchers wishing to target their CAD tools to industrial FPGAs can obtain a full list of the legality constraints in Altera FPGAs from the Quartus University Interface Program [2].

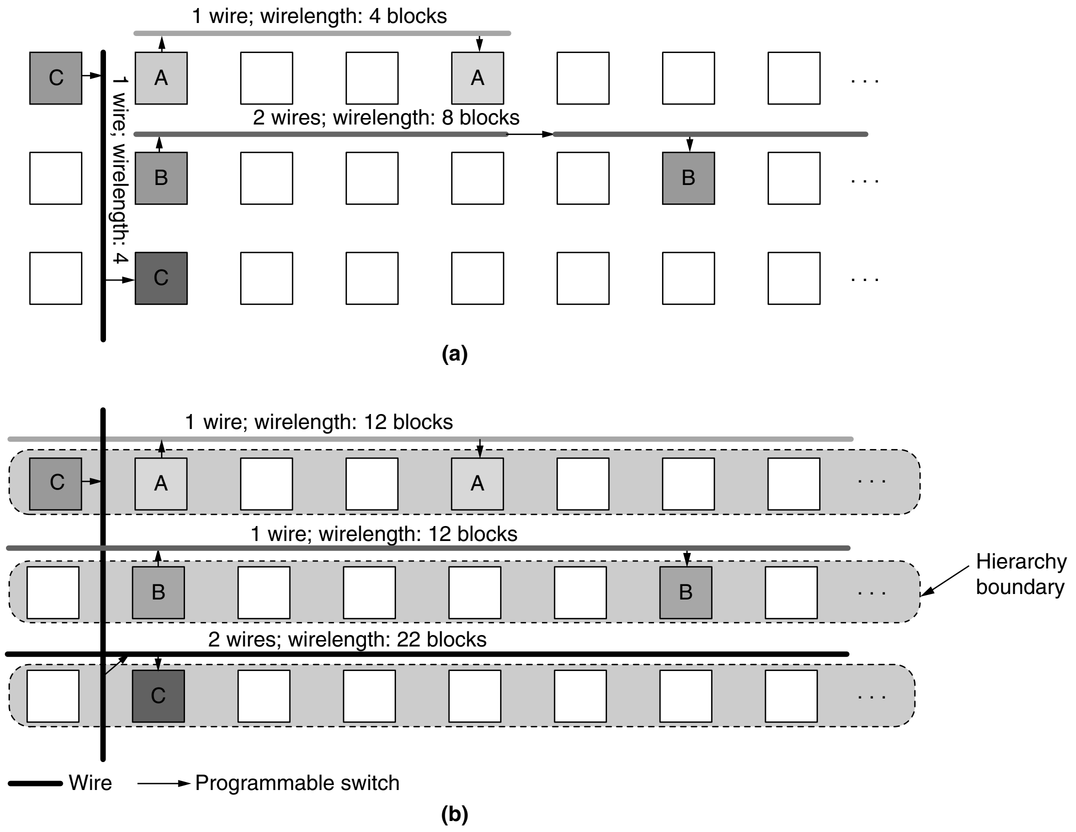

1 wire; wirelength: 4 blocks
C
A
A
1 wire; wirelength: 4
2 wires; wirelength: 8 blocks
B
B
C
(a)
1 wire; wirelength: 12 blocks
C
A
A
1 wire; wirelength: 12 blocks
Hierarchy boundary
B
B
2 wires; wirelength: 22 blocks
C
Wire
Programmable switch
(b)

FIGURE 14.2 ■ Influence of the routing architecture on wirelength for a given placement: (a) sample routings on a Stratix-II FPGA (island style), and (b) sample routings on an APEX FPGA (hierarchical).

minimize not only the total wiring required by the design but also the amount of *routing congestion*. Routing congestion occurs when the interconnect demand approaches or exceeds the fabricated wiring capacity in some part of the FPGA.

In addition to optimizing for routability, timing-driven algorithms use timing analysis [5] to identify critical paths and/or connections and to optimize the delay of those connections. Since most delays in an FPGA are due to the programmable interconnect, timing-driven placement can achieve a large improvement in circuit speed over routability-driven approaches.

Some recent FPGA placement algorithms attempt to minimize power consumption as well.

14.1.3 Designer Placement Directives

Commercial FPGA placement tools allow designers to control the placement of some or all of the design logic at various levels of abstraction. Obeying the placement directives specified by a designer while still choosing good locations for the unconstrained and partially constrained blocks is a challenging problem, but one on which little has been published.

Figure 14.3 illustrates the common types of placement directives. The most restrictive specifies the *exact location* of a block. Typical uses of this directive are to lock down the design I/Os at the locations required by the circuit board or to lock down the elements of a performance-critical intellectual property (IP) core. A less restrictive directive forces blocks to go into a specific two-dimensional area, or *fixed region*. This directive allows a designer to guide the placement tool to a good high-level floorplan while still allowing automatic optimization of the placement details. One can specify the *relative location* of several blocks, but let the placement tool choose exactly where to locate the block group. This directive is useful for library components where a designer knows a good placement of the component blocks relative to each other. A *floating region* specifies that some logic should be placed within a tight region but that the placement tool can choose where that region should be on the device.

One must take care when specifying placement directives, as fixing portions of the placement ineffectively will reduce result quality versus a fully automatic placement. Modern placement tools produce high-quality results, and generally

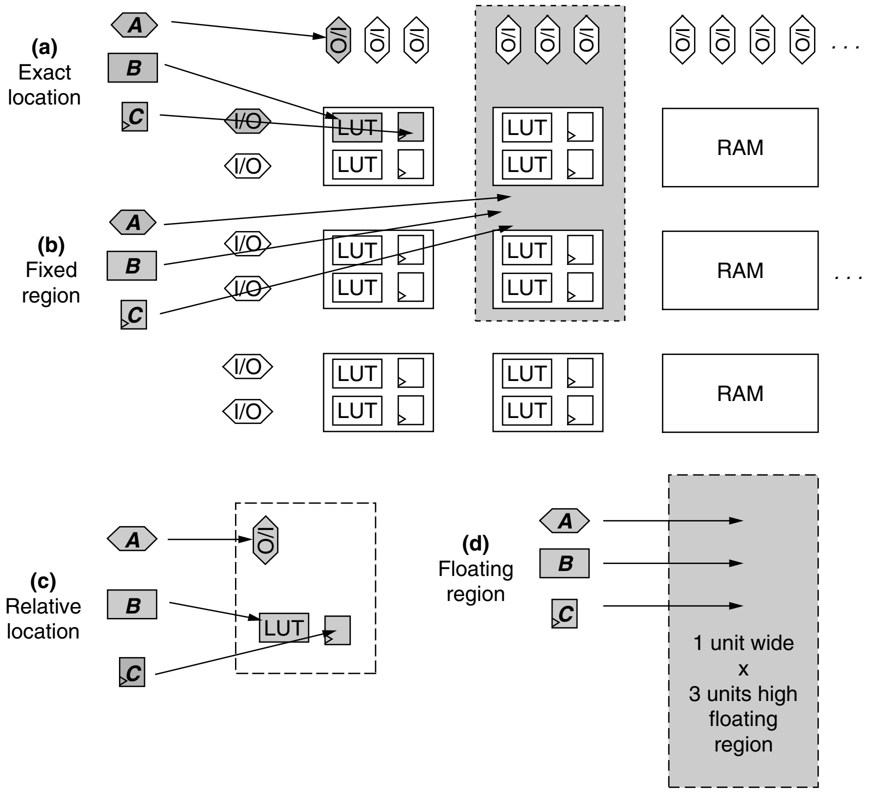


FIGURE 14.3 ■ Placement directives, ordered from most to least restrictive: (a) exact location, (b) fixed region, (c) relative location, and (d) floating region.

it is very difficult for a designer to specify placement directives on irregular logic that lead to a better solution than the placement tool would find without guidance. Placement directives have more value for regular structures, since humans are better than conventional CAD tools at recognizing regular logic patterns and matching them to a highly optimized regular placement. For examples of the use of placement directives, see Chapter 16.

14.2 CLUSTERING

A common companion to FPGA placement algorithms is a bottom-up clustering step that runs before the main placement algorithm to group-related circuit elements together into clusters. Clustering reduces the number of blocks to place, improving the runtime of the main placement algorithm. In addition, one normally chooses a cluster size that corresponds to a natural boundary in the FPGA architecture, such as a logic block. This allows the clustering algorithm to deal with many of the device legality constraints by ensuring that each cluster forms a legal logic (or RAM or DSP) block, and it simplifies legality checking for the main placement algorithm.

The most common FPGA clustering formulation transforms a netlist of logic elements into a netlist of logic blocks. In most FPGA architectures each logic element consists of a LUT plus a register, and each logic block has the capacity to implement up to *N* logic elements. As well, logic blocks have a limit on the number of input signals that can be brought in from the programmable routing and on the number of different control signals, such as register clocks, that can be used.

The typical clustering goals are:

- To achieve high density by minimizing the number of clusters (i.e., logic blocks) required to implement a circuit.
- To improve circuit speed by localizing time-critical connections within a cluster so they can be completed with fast local routing.
- To reduce wiring demand in the FPGA by grouping related logic in each cluster.

The RASP system [6] includes one of the first logic block clustering algorithms. It performs maximum weighted matching on a graph where edge weights between logic elements reflect the desirability of clustering them. Logic elements that cannot be legally clustered have no edge between them, while those connected by timing-critical connections or with a large number of common signals have edges with high weights.

RASP has the attractive feature of simultaneously choosing all clusters of two logic elements to maximize the total weight of edges contained within the clusters. By recursively repeating the algorithm, one can create larger clusters, at least when the cluster capacity is a power of 2. The first matching produces a netlist of size-2 clusters; a matching on the size-2 cluster netlist produces size-4 clusters, and so on. The RASP clustering algorithm has a high computational complexity of $O(n^3)$, where n is the number of logic elements in the circuit. This prevents it from scaling to large problems.

The VPack algorithm [3] takes the opposite approach to that of RASP—it creates one cluster of the desired size (e.g., seven logic elements) before moving on to create the next cluster. VPack first chooses a *seed* logic element for a new cluster and then greedily packs the logic element with the highest *attraction* to the current cluster until no more can be legally added. The attraction function is the number of nets that connect to both the logic element in question and the current cluster. VPack has a computational complexity of $O(k_{\max} n)$ where $k_{\max}$ is the maximum fanout of any net in the design, so it scales well to large problems.

Many algorithms that use the same basic procedure as VPack, but different attraction functions, have been published. The T-VPack algorithm by Marquardt et al. [3, 7] is a timing-driven enhancement of VPack where the attraction function for a logic element, L, to cluster C becomes

$$Attraction(L) = 0.75 \cdot \sum_{j \in conn(L,C)} criticality(j) + 0.25 \cdot \frac{|Nets(L) \cap Nets(C)|}{MaxNets} \tag{14.1}$$

The first term in equation 14.1 gives higher attraction to logic elements that are connected to the current cluster by timing-critical connections, while the second term is taken from VPack and favors grouping together logic elements with many common signals. To find the criticality of each connection, a timing analysis is performed with a simple delay model to determine each connection's timing *slack*. The slack of a connection [5] is defined as the amount of delay that can be added to that connection before some path through it limits the circuit speed. The *criticality* of a connection, j, is then given by

$$criticality(j) = 1 - \frac{slack(j)}{D_{\max}} \tag{14.2}$$

where $D_{\max}$ is the delay of the longest path in the circuit. Connections on the critical path (i.e., with no timing slack) have a criticality of 1, while connections with a large amount of slack have a criticality near 0.

Somewhat surprisingly, T-VPack improves not only circuit speed over VPack but also reduces the amount of programmable routing required between clusters. By absorbing more connections within clusters, T-VPack is able to capture more nets entirely within a cluster, which reduces wiring demand between logic blocks.

The iRAC [8] clustering algorithm uses an attraction function that favors the absorption of small nets within a cluster:

$$Attraction(L, C) = \sum_{i \in Nets(L) \cap Nets(C)} k(i, L, C) \cdot \frac{[1 + pins\_in\_cluster(i, C)]}{|pins(i)|}$$

$$k(i, L, C) = \begin{cases} 10, \text{ if adding } L \text{ to } C \text{ would absorb net } i \text{ within } C \\ 1, \text{ otherwise} \end{cases} \tag{14.3}$$

The attraction function (equation 14.3) weights nets more heavily with a small number of terminals outside the cluster, and also gives a ten-times attraction bonus to any net that would be immediately absorbed by adding block L to the cluster. By reducing the number of nets to be routed between logic blocks, iRAC achieves an improvement in routability over T-VPack.

Lamoureaux and Wilton [9] have developed a power-aware enhancement of T-VPack. They modify equation 14.1 by adding a power minimization term that weights each connection from block L to cluster C by its *switching activity*. The switching activity of a signal is the number of times it is expected to change state per second. The power minimization term favors the absorption of nets that frequently switch logic states, resulting in lower capacitance for these nets and lower overall dynamic power.

14.3 SIMULATED ANNEALING FOR PLACEMENT

Simulated annealing is the most widely used placement algorithm for FPGAs. It mimics the annealing procedure by which strong metal alloys are created—initially blocks can move fairly freely, but as the *temperature* drops they gradually freeze into a high-quality placement [10].

Figure 14.4 shows the basic flow of simulated annealing for placement. First an initial placement is generated. This initial placement is generally of low quality, and is often created simply by assigning each block to the first legal location found. The placement is then iteratively improved by proposing and evaluating placement perturbations, or *moves*. A placement perturbation is proposed by a *move generator*, generally by moving a small number of blocks to new locations. A *cost function* is used to evaluate the impact of each proposed move.

Moves that reduce cost are always accepted, or committed to the placement, while those that increase cost are accepted with probability

$$e^{-\frac{\Delta Cost}{T}}$$

where T is the current *temperature*. This function ensures that moves that increase the cost by an amount that is small compared to the current temperature are likely to be accepted, while moves that increase the cost by an amount much larger than the current temperature are not. Accepting some moves that increase the cost helps escape local minima and produces a higher-quality final placement. At the start of the anneal, temperature is high; it gradually decreases according to the *annealing schedule*. This schedule also controls how many moves are performed between temperature updates and when the placement is considered sufficiently optimized that the anneal should end.

Two key strengths of simulated annealing that make it well suited to FPGA placement are:

1. One can enforce all the legality constraints imposed by the FPGA architecture fairly directly. The two basic techniques are to forbid the creation of illegal placements in the move generator or to add a penalty cost to illegal placements.

```
P = InitialPlacement ();
T = InitialTemperature ();

while (ExitCriterion () == False) {
    while (InnerLoopCriterion () == False) { /* One temperature */
         $P_{new}$ = PerturbPlacementViaMove (P);
         $\Delta$Cost = Cost ($P_{new}$) - Cost (P);
         r = random (0,1);
         if (r < $e^{-\Delta Cost/T}$) {
            P = $P_{new}$ ; /* Accept move */
         }
     }  /* End one temperature */
     T = UpdateTemp (T);
}
```

FIGURE 14.4 ■ Pseudo-code of a generic simulated annealing placement algorithm. (*Source*: Adapted from [13].)

2. By creating an appropriate cost function, one can directly model the impact of the FPGA routing architecture on circuit delay and routing congestion.

14.3.1 VPR and Related Annealing Algorithms

VPR [3,11,12] is a popular timing-driven simulated annealing placement tool. It is usually used in conjunction with T-VPack, or a similar clustering algorithm, that preclusters the logic elements into legal logic blocks. One of VPR's main features is that it can automatically adapt to different FPGA architectures so long as they employ island-style routing.

VPR's annealing schedule is based on parameters computed during placement rather than on fixed starting and ending temperatures and a fixed cooling rate. This adaptive annealing schedule generates high-quality results across a wide range of design sizes, FPGA architectures, and cost functions, making it preferable to more "hardcoded" schedules. VPR sets the *InitialTemperature* to 20 times the cost change of the average move, and the *ExitCriterion* is met when the temperature is less than 0.5 percent of the cost divided by the number of nets in the circuit. The fraction of moves that are accepted at each temperature, α, is monitored throughout the anneal.

Lam and Delosme [14] showed that simulated annealing makes the largest improvements to a placement when α is near 44 percent. Consequently, VPR rapidly decreases the temperature when α is significantly above or below 44 percent and slowly decreases it when α is near 44 percent in order to spend the majority of the annealing time in the most productive range. The move generator used by VPR to find placement perturbations also varies as the anneal progresses in order to keep α near 44 percent. When a block is picked for a move, its new proposed location will always be within a window with a Manhattan radius of *range limit* blocks. Initially, the range limit is the size of the entire chip, allowing a block to move anywhere in the device in one move.

As the anneal progresses, the range limit shrinks so that the moves proposed are smaller local improvements, since these are the most likely moves to be accepted as the placement converges to an increasingly high-quality solution. More specifically, whenever the temperature is updated in Figure 14.4, VPR also updates the range limit according to

$$range\_limit\,(new) = range\_limit\,(old) \cdot (1 - 0.44 - \alpha) \tag{14.4}$$

VPR's cost function [12] also has some ability to adapt to different FPGA architectures:

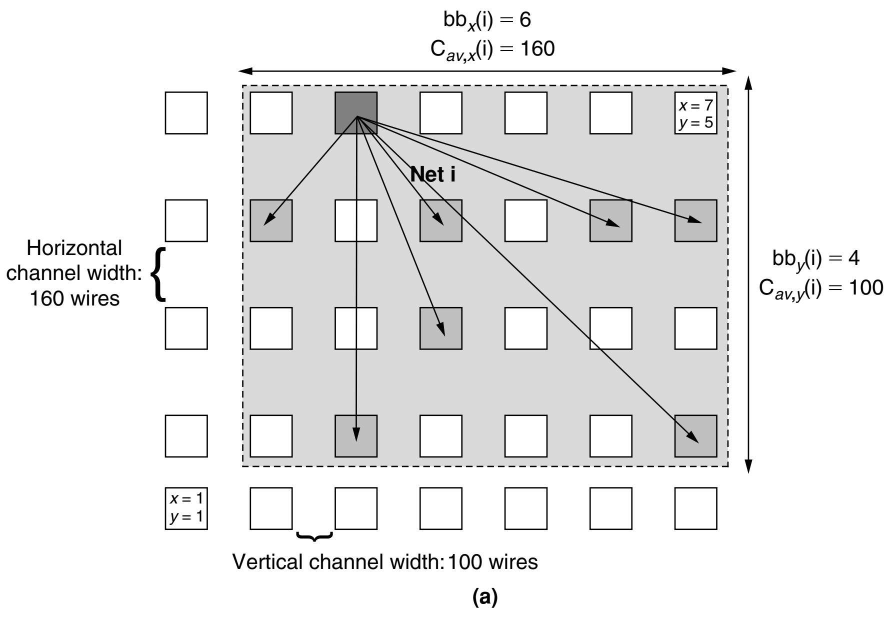
$$\begin{aligned} Cost = (1-\lambda) \sum_{i \in AllNets} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)} + \frac{bb_y(i)}{C_{av,y}(i)} \right] \\ + \lambda \sum_{j \in AllConnections} Criticality(j) \cdot Delay(j) \end{aligned} \tag{14.5}$$

The first term in equation 14.5 causes the placement algorithm to optimize an estimate of the routed wirelength, normalized to the average wiring capacity in each region of the FPGA. The wirelength needed to route each net *i* is estimated as the bounding box span (bb_x and bb_y) in each direction, multiplied by a fanout-based correction factor, $q(i)$. As Figure 14.5(a) illustrates, the bounding box of a net is simply the smallest rectangle that encloses all the net terminals. Figure 14.5(b) shows that for higher fanout nets, the bounding box span underpredicts the wiring needed. For the eight-terminal net shown, the sum of bb_x and bb_y is 10 units, but even a best-case routing requires 11 units of wire. $q(i)$ is 1 for two- and three-terminal nets and slowly increases with net terminal count to compensate for this underprediction [16].

The corrected bounding box span is a reasonable estimate of the routed wirelength for an island-style FPGA that contains at least some short wiring segments that span only a few logic blocks. Most recent commercial FPGAs, including the Altera Stratix and Xilinx Virtex [15] families, meet this condition. Equation 14.5 does not contain a good estimate of wirelength for other FPGA types, such as hierarchical FPGAs, so this cost function would not perform well with them.

Some FPGAs have differing amounts of routing available in the vertical direction compared to the horizontal direction, or in different regions of the chip. For example, a Stratix-II FPGA has 1.6 times as much horizontal as vertical routing, and some routing is not available over the large 576-kbit RAM blocks. Therefore, the routing capacity is not uniform everywhere in the device. In such cases, it is beneficial to move wiring demand to the more routing-rich direction or regions. Accordingly, the cost function of equation 14.5 scales the estimated wiring in each direction by the average routing capacity over the net bounding box in that direction. Figure 14.5(a) shows an example computation.

The second term in equation 14.5 optimizes timing by favoring placements in which timing-critical connections have the potential to be routed with low delay. To evaluate the second term quickly, VPR needs to be able to rapidly estimate the delay of a connection. It makes use of the fact that the delay between two points in an island-style FPGA is primarily a function of the distance between them. Before placement begins, VPR precomputes a table of best-case routing



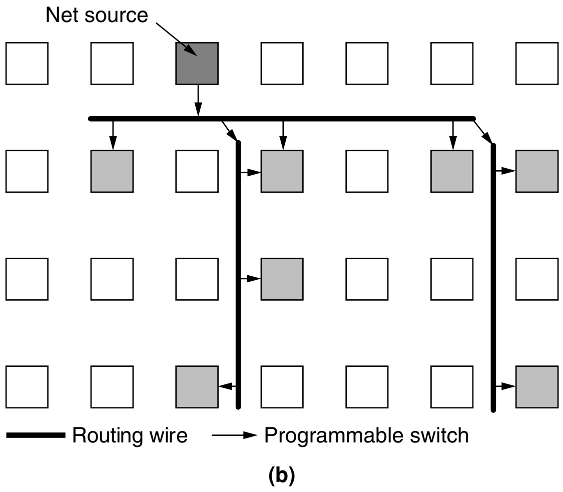


FIGURE 14.5 ■ An example wirelength cost computation: (a) net bounding box and average channel capacity; (b) best-case routing, with a wirelength of 11.

delays for every possible distance between pairs of points. The delay table entries are computed by invoking a router with each possible $(\Delta x, \Delta x)$—the router finds the fastest path between the two endpoints.

Periodically (generally once per temperature) VPR computes the delay of every connection given the current placement and then performs a timing analysis to find each connection's slack. Equation 14.2 computes the criticality of each connection given its slack. Consequently, VPR's estimate of which connections are critical changes as placement progresses, and timing optimization can move from one part of the circuit to another.

One of the important features of VPR's cost function is that, with appropriate coding, the cost change caused by the motion of a constant number of blocks can be computed in constant time. This enables many moves to be evaluated during the placement of a large circuit, which is one of the keys to obtaining a high-quality placement with simulated annealing. The overall computational complexity of VPR is $O(n^{1.33})$ [3], where n is the number of functional blocks to be placed, allowing VPR to scale well to large circuits.

Many enhancements have been made to the original VPR algorithm. The PATH algorithm by Kong [17] uses a new timing criticality formulation in which the criticality of a connection is a function of the slacks of all the paths passing through it, rather than just a function of the worst (smallest) slack. This technique increases the cost function weighting on connections with many critical or near-critical paths, which is beneficial because a move that reduces the delay of such a connection can improve many important timing paths simultaneously. On average, PATH reduces critical path delay by 15 percent compared to VPR.

The SCPlace algorithm [18] enhances VPR so that a portion of the moves are *fragment moves* in which a single logic element is moved instead of an entire logic block. This allows the placement algorithm to modify the initial clustering to shorten connections that are now seen to be poorly localized. Fragment moves improve both circuit timing and wirelength.

Sankar and Rose [19] explored a trade-off between reduced result quality and extremely low placement runtimes. Instead of simply clustering logic elements into logic blocks, their *hierarchical annealing* algorithm clusters logic blocks twice into larger units, as shown in Figure 14.6. The first-level clustering creates

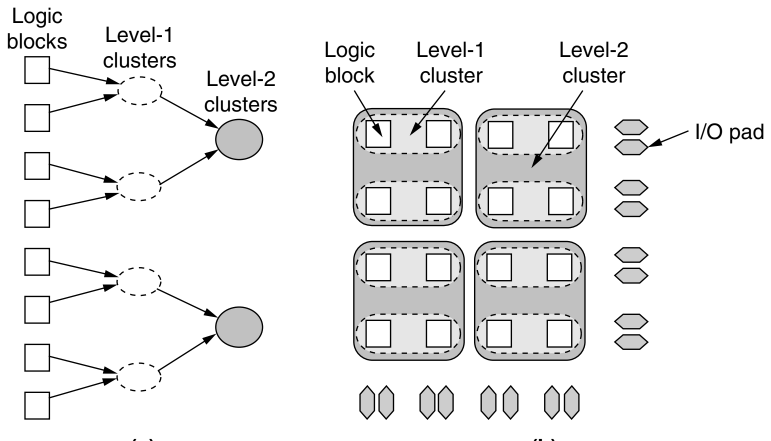


FIGURE 14.6 ■ An overview of hierarchical annealing: (a) multilevel clustering, and (b) placement of large clusters followed by unclustering and placement refinement.

clusters that each contain approximately 64 logic blocks, and the second-level clustering groups four level-1 clusters into each level-2 cluster. Placement of a netlist of level-2 clusters is very fast because there are relatively few blocks to place. To make placement of the level-2 clusters even faster, Sankar and Rose [19] use a greedy (temperature = 0 anneal) iterative improvement algorithm, seeded with a fast constructive (instead of random) placement. Once placement of the level-2 clusters is complete, a level-1 initial placement is created by locating each level-1 cluster inside the boundary of the level-2 cluster that contained it.

The placement of level-1 clusters is refined by a temperature-0 anneal. The clusters are then replaced by their constituent logic blocks and the placement of each logic block is fine-tuned with a *low-temperature* anneal. The initial temperature for this anneal is selected so that only moves that reduce cost or increase it a small amount are allowed; consequently, the initial placement solution has a large impact on the final placement. For very fast CPU times this algorithm significantly outperforms VPR in achieved wirelength, but it lags behind VPR for longer permissible CPU times.

Lamoureaux and Wilton [9] modified VPR's cost function by adding a third term, *PowerCost*, to equation 14.5.

$$PowerCost = \sum_{i \in AllNets} q(i)\left[bb_x(i) + bb_y(i)\right] \cdot Activity(i) \tag{14.6}$$

where *Activity(i)* is the average number of times net *i* transitions per second. This additional cost term reduces circuit power by focusing more effort on localizing rapidly transitioning nets.

14.3.2 Simultaneous Placement and Routing with Annealing

Instead of relying on fast heuristics to estimate placement routability and timing, some algorithms use a router to obtain a partial or complete routing for each placement proposed during the anneal. These algorithms can directly extract wiring usage, congestion, and timing from the circuit routing, so their cost functions can be very detailed. Another of their advantages is that one can develop a placement algorithm that automatically adapts to a wider class of FPGA architectures, since fewer (or ideally no) assumptions about the device-routing architecture need to be incorporated into the cost function. The disadvantage of using a router in the cost function is CPU time. Evaluating the cost change after each move is very CPU intensive, making it difficult to evaluate enough moves to obtain high-quality placements for large circuits in a reasonable time.

PROXI [20] is a timing-driven FPGA placement algorithm that uses a router to compute its cost function. The PROXI cost function is a weighted sum of the number of unrouted nets and the delay of the circuit critical path. After each placement perturbation, PROXI rips up all of the nets connected to blocks that have moved and reroutes them via a fast, directed-search maze router [21].

To improve CPU time, PROXI allows the maze router to explore only a small portion of the routing fabric at high temperatures—if no unblocked routing path is found quickly, the net is left unrouted. At lower temperatures, the placement is of higher quality and the router is allowed to explore a larger portion of the routing fabric. After each net is rerouted, the critical path is recomputed incrementally. PROXI produces high-quality results, but requires high CPU time.

Independence [22] is an FPGA placement tool that can effectively target a wide variety of FPGA routing architectures. It is purely routability-driven, and its cost function monitors both the amount of wiring used by the placement and the routing congestion:

$$\begin{aligned} Cost = \sum_{i \in Nets} RoutingResources(i) + \lambda \\ \sum_{k \in RoutingResources} \max(Occupancy(k) - Capacity(k), 0) \end{aligned} \tag{14.7}$$

The λ parameter in equation 14.7 is a heuristic weighting factor. Independence uses the PathFinder routing algorithm [23] to find new routes for all affected nets after each move. Instead of leaving nets unrouted when there is no unblocked path, PathFinder allows *wire congestion* by routing two nets on the same routing resource. Such a routing is not legal; however, by summing the overuse of all the routing resources in the FPGA, Independence can directly monitor the amount of routing congestion implicit in the current placement. The Independence cost function monitors not only routing congestion but also the total wirelength used by the router to create a smoother cost function that is easier for the annealer to optimize. Independence produces high-quality results on a wide variety of FPGA architectures, including both island style and hierarchical, but it requires very high CPU time.

14.4 PARTITION-BASED PLACEMENT

Another popular placement approach recursively partitions the circuit netlist and assigns each partition to a different physical region in the FPGA. Usually each partitioning step divides a previous (larger) partition into two pieces, or *bipartitions* the component, although some algorithms perform *multiway partitioning* to produce a larger number of circuit partitions in each step. Partitioning algorithms attempt to minimize the number of nets that are cut, or that cross, between partitions. Since each partition of the circuit will be assigned to a different region of the FPGA, partition-based placement minimizes the number of nets leaving each region and hence indirectly optimizes the amount of wiring required by the design. Partition-based placement can leverage the availability of high-quality, CPU-efficient partitioning algorithms, making this approach scalable to large problems. However, for some FPGA architectures, partition-based placement suffers from the disadvantage that it does not directly optimize the circuit timing or the amount of routing required by the placement.

Hierarchical FPGAs are good candidates for partition-based placement, since their routing architectures create natural partitioning cut lines. Hutton et al. [24] describe a commercial placement algorithm for the Altera Apex 20K family that recursively partitions the circuit along the cut lines formed by the routing hierarchy, as shown in Figure 14.7. This algorithm is made timing-driven by heavily weighting connections with low slack during each partitioning phase and by partitioning to minimize weighted cut size. This encourages partitioning solutions in which timing-critical connections can be routed using the fast routing available within the lower levels of the routing hierarchy. To improve the prediction of the critical path, the delay estimate for each connection is a function of (1) the number of hierarchy boundaries the net must traverse because of the known partition cuts at the higher levels of the routing hierarchy, and (2) statistical estimates of how many hierarchy boundaries the connection will cross at future partitioning steps.

Recursive partitioning has also been used for placement in island-style FPGAs. ALTOR [25] was originally developed for standard cell circuits, but was adapted to FPGAs and widely used in FPGA research. Figure 14.8 shows the sequence of cut lines used by ALTOR to target an island-style FPGA—note that the sequence is quite different from that used with a hierarchical FPGA. In an island-style FPGA, blocks separated by a short Manhattan distance can be connected with a small amount of routing. Consequently, the cut lines are designed to divide the FPGA into ever-shrinking squares—the fewer signals that must leave each square, the less interconnect required.

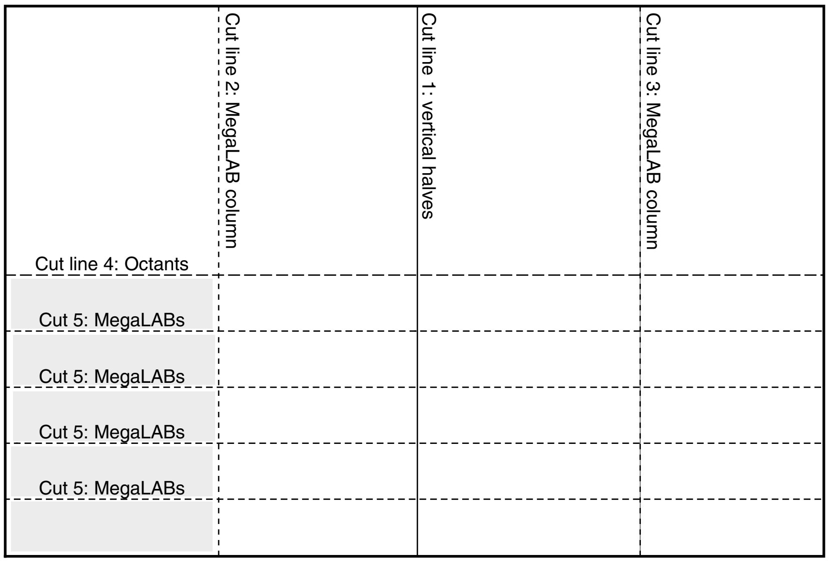

Cut line 2: MegaLAB column
Cut line 1: vertical halves
Cut line 3: MegaLAB column
Cut line 4: Octants
Cut 5: MegaLABs
Cut 5: MegaLABs
Cut 5: MegaLABs
Cut 5: MegaLABs

FIGURE 14.7 ■ The partitioning sequence for the APEX 20K FPGA.

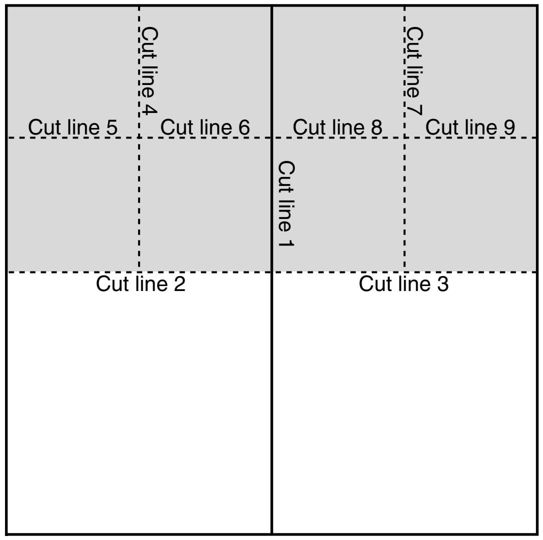


FIGURE 14.8 ■ The partitioning sequence for an island-style FPGA.

ALTOR's first cut line divides the chip into two halves vertically. The second cut line divides the left half of the circuit into upper left and lower left quarters. The third cut line divides the right half of the circuit in the same way. When partitioning along the third cut line, ALTOR uses *terminal propagation* [26] from the left half of the chip, which is already partitioned into an upper and lower quarter, to bias the partitioning of the right half. For example, the net shown in Figure 14.9 has one terminal in the right half of the chip and one terminal in the upper left corner. During partitioning along cut line 3, this net is considered to have a fixed terminal in the upper partition, which will bias the partitioner to keep the free terminal of this net in the partition above cut line 3. Terminal propagation reduces final wirelength by optimizing the placement of the terminals of nets that have been cut in some partitioning step.

Maidee et al. [27] developed a timing-driven placement algorithm for island-style FPGAs that employs both partitioning and annealing. Before partitioning begins, the VPR router is used to generate a table of net delay versus distance spanned by the net that takes into account the FPGA routing architecture. As partitioning proceeds, the algorithm records the minimum length each net can achieve given the current number of partitioning boundaries it crosses. The delay corresponding to each net's span is retrieved from the net delay versus span table, and a timing analysis is performed to identify critical connections.

Timing-critical connections to terminals outside the region being partitioned act as anchor points during each partitioning. This forces the other end of the connection to be allocated to the partition that allows the critical connection to be short. Once partitioning has proceeded to the point that each region contains only a few cells, any overfilled regions are legalized with a greedy movement

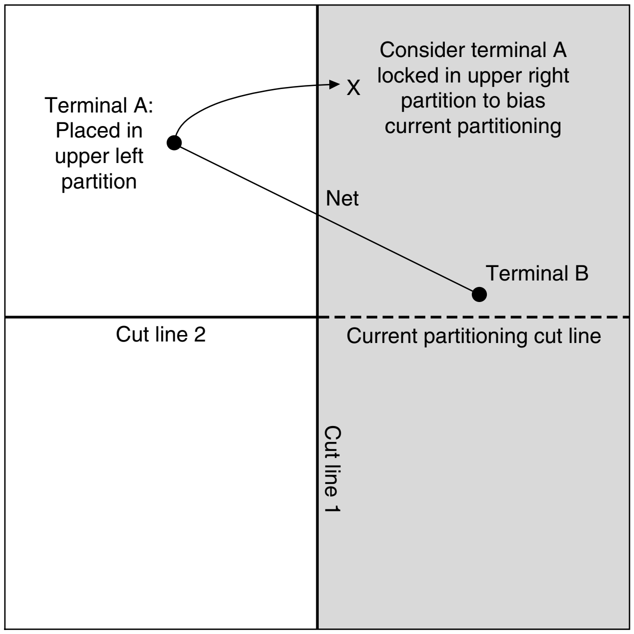


FIGURE 14.9 ■ An example of terminal propagation.

heuristic. Finally, the VPR annealing algorithm is invoked with a low starting temperature to "fine-tune" the placement. This fine-tuning step allows blocks to move anywhere in the device, so early placement decisions made by the partitioner, when little information about the critical paths or the final wirelength of each net was available, can be reversed. This algorithm achieves wirelength and speed results comparable to those of a full VPR anneal, with significantly reduced CPU time.

14.5 ANALYTIC PLACEMENT

Analytic algorithms are based on creating a smooth function of a placement that approximates routed wirelength. Efficient numerical techniques are used to find the global minimum of this function; if the function approximates wirelength well, this solution is a placement with good wirelength. However, this global minimum is usually an illegal placement, so constraints and heuristics must be applied to guide the algorithm to a legal solution.

While analytic placement approaches are popular for ASICs, few exist for FPGAs, likely due to the more difficult FPGA placement legality constraints. The Negotiated Analytic Placement (NAP) algorithm from Chan and Schlag [28] targets FPGAs and has several novel features, including some that make it suitable for implementation on multiple processors in parallel.

14.6 FURTHER READING AND OPEN CHALLENGES

While this chapter has focused on placement algorithms specifically designed for FPGAs, there is also a great deal of literature on placement for custom-manufactured integrated circuits, much of which is relevant to FPGAs. For a recent overview of general placement algorithms, see Cong et al. [29]. This chapter also treated placement as separate from synthesis. Recent commercial and academic tools incorporate *physical synthesis*, however, where portions of the circuit are resynthesized as placement proceeds and more information about critical paths becomes available. For an overview of FPGA physical synthesis and its interaction with placement, see Hutton and Betz [13].

The greatest challenge facing FPGA placement is the need to produce high-quality placements for ever-larger circuits. FPGA capacity doubles every two to three years, doubling the size of the placement problem at the same rate. In addition, uniprocessor speed is no longer increasing as quickly as it did in the past, which means that single processor speed will increase by less than two times in the same period. In order to maintain the fast time to market and ease of use historically provided by FPGAs, placement algorithms cannot be allowed to take ever more CPU time. There is thus a compelling need for algorithms that are very scalable yet still produce high-quality results.

The roadmap for future microprocessors indicates that the number of independent processors, or cores, on a single chip will increase rapidly in the coming years. Consequently, most engineers will have parallel computers on their desktops. Part of the solution to the problem of keeping FPGA placement times reasonable may be to find techniques and algorithms to exploit parallel processing without sacrificing result quality.

References

[1] D. Lewis, E. Ahmed, G. Baeckler. The Stratix-II routing and logic architecture. *Proceedings of the 13th ACM International Symposium on Field-Programmable Gate Arrays,* 2005.

[2] The Quartus University Interface Program (*www.altera.com/education/univ/research/unv-quip.html*).

[3] V. Betz., J. Rose, A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer, February 1999.

[4] R. Cliff, et al. A next generation architecture optimized for high density system level integration. *Proceedings of the 21st IEEE Custom Integrated Circuits Conference*, 1999.

[5] R. Hitchcock, G. Smith, D. Cheng. Timing analysis of computer hardware. *IBM Journal of Research and Development*, January 1983.

[6] J. Cong, J. Peck, Y. Ding. RASP: A general logic synthesis system for SRAM-based FPGAs. *Proceedings of the Fifth International Symposium on Field-Programmable Gate Arrays*, 1996.

[7] A. Marquardt, V. Betz, J. Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. *Proceedings of the Seventh International Symposium on Field-Programmable Gate Arrays*, 1999.

[8] A. Singh, M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2002.
[9] J. Lamoureaux, S. Wilton. On the interaction between power-aware FPGA CAD algorithms. *Proceedings of the International Symposium on Computer-Aided Design*, 2003.
[10] S. Kirkpatrick, C. Gelatt, M. Vecchi. Optimization by simulated annealing. *Science* 2(20), May 1983.
[11] V. Betz, J. Rose. VPR: A new packing, placement and routing tool for FPGA research. *Proceedings of the Seventh International Conference on Field-Programmable Logic and Applications*, 1997.
[12] A. Marquardt, V. Betz, J. Rose. Timing-driven placement for FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2000.
[13] M. Hutton, V. Betz. *Electronic Design Automation for Integrated Circuits Handbook*, Taylor and Francis, eds. (Chapter 13), CRC Press, 2006.
[14] J. Lam, J. Delosme. Performance of a new annealing schedule. *Design Automation Conference*, 1988.
[15] *Virtex Family Datasheet* (*www.xilinx.com*).
[16] C. Cheng. RISA: Accurate and efficient placement routability modeling. *Proceedings of the International Conference on Computer-Aided Design*, 1994.
[17] T. Kong. A novel net weighting algorithm for timing-driven placement. *Proceedings of the International Conference on Computer-Aided Design*, 2002.
[18] G. Chen, J. Cong. Simultaneous timing driven clustering and placement for FPGAs. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2004.
[19] Y. Sankar, J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1999.
[20] S. K. Nag, R. A. Rutenbar. Performance-driven simultaneous placement and routing for FPGAs. *IEEE Transactions on Computer-Aided Design*, June 1998.
[21] Y. C. Lee. An algorithm for path connections and applications. *IRE Transactions on Electronic Computing*, September 1961.
[22] A. Sharma, C. Ebeling, S. Hauck. Architecture-adaptive routability-driven placement for FPGAs. *Proceedings of the International Symposium on Field-Programmable Logic and Applications*, 2005.
[23] L. McMurchie, C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. *Proceedings of the Fifth International Symposium on Field-Programmable Gate Arrays*, 1995.
[24] M. Hutton, K. Adibsamii, A. Leaver. Adaptive delay estimation for partitioning-driven PLD placement. *IEEE Transactions on VLSI* 11(1), February 2003.
[25] J. Rose, W. Snelgrove, Z. Vranesic. ALTOR: An automatic standard cell layout program. *Proceedings of the Canadian Conference on VLSI*, January 1985.
[26] A. Dunlop, B. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, January 1985.
[27] M. Maidee, C. Ababei, K. Bazargan. Fast timing-driven partitioning-based placement for island style field-programmable gate arrays. *Design Automation Conference*, 2003.
[28] P. Chan, M. Schlag. Parallel placement for field-programmable gate arrays. *Proceedings of the 11th International Symposium on Field-Programmable Gate Arrays*, 2003.
[29] J. Cong, J. Shinnerl, M. Xie, T. Kong, X. Yuan. Large-scale circuit placement. *ACM Transactions on Design Automation of Electronic Systems*, April 2005.