PATHFINDER: A NEGOTIATION-BASED, PERFORMANCE-DRIVEN ROUTER FOR FPGAS

Larry McMurchie
Synplicity Corporation
Carl Ebeling
Department of Computer Science and Engineering
University of Washington

Routing is a crucial step in the mapping of circuits to field-programmable gate arrays (FPGAs). For large circuits that utilize many FPGA resources, it can be very difficult and time consuming to successfully route all of the signals. Additionally, the performance of the mapped circuit depends on routing critical and near-critical paths with minimum interconnect delays. One disadvantage of FPGAs is that they are slower than their ASIC counterparts, so it is important to squeeze out every possible nanosecond of delay in the routing.

The first goal, a complete routing of all signals, is difficult to achieve in FPGAs because of the hard constraints on routing resources. Unlike ASICs and printed circuit boards (PCBs), FPGAs have a fixed amount of interconnect. The usual approach in placement is to minimize the wiring resources anticipated for routing signals. Although this reduces the overall demand for resources, signals inevitably compete for the same resources during routing. The challenge is to find a way to allocate resources so that all signals can be routed. The second goal, minimizing delay, requires the use of minimum-delay routes for signals, which can be expensive in terms of routing resources, especially for high-fanout signals. Thus, the solution to the entire routing problem requires the simultaneous solution of two interacting and often competing subproblems.

Early solutions to the FPGA routing problem were based on the considerable literature on routing in the context of ASICs and gate arrays. The problem of routing FPGAs bears a considerable resemblance to the problem of global routing for custom integrated circuit design, where signals are assigned to channels. However, the two problems differ in several fundamental respects. First, routing resources in FPGAs are discrete and scarce while they are relatively continuous in custom integrated circuits (ICs). For this reason FPGAs require an integrated approach using both global and detailed routing. A second difference is that global routing for custom ICs is based on an undirected graph embedded in Cartesian space (i.e., a two-dimensional grid). In FPGAs the switches are often directional, and the routing resources connect arbitrary (but fixed) locations, requiring a directed graph that may not be embedded in Cartesian space. Both of these distinctions are important, as they prevent direct application of much of the previous work in routing.

By far, the most common approach to global routing of custom ICs is a shortest-path algorithm with obstacle avoidance. By itself, this technique usually yields many unroutable nets that must be rerouted by hand. A plethora of rip-up and retry approaches have been proposed to remedy this deficiency [1–3]. The basic problem with rip-up and retry is that the success of a route is dependent not just on the choice of nets to reroute but also on the order in which rerouting is done. Delay is usually factored into the standard rip-up and retry approach by ordering the nets to be routed such that critical nets are routed most directly [4–6].

To make the FPGA routing problem tractable, nearly all of the routing schemes in the literature incorporate features of the underlying architecture. Palczewski [7] describes a maze router with rip-up and reroute targeting the Xilinx 4000 series. In this work the structure of the plane-parallel switchbox in the 4000 series is exploited in conjunction with an A* search. Brown et al. [4] employ an architecture model consisting of channels, switchboxes, connection matrices, and logic blocks. A global router balances channel densities and a detailed router generates families of explicit paths within channels to resolve congestion. These approaches, as well as others, obtain some of their success by exploiting the features of a particular architecture model. The problem is that new architectures become constrained by the restrictions of such existing routing algorithms.

17.1 THE HISTORY OF PATHFINDER

PathFinder was used initially in the development of the Triptych FPGA architecture [8–10]. In fact, Triptych, with its heavy reliance on effective placement and routing tools, was a catalyst for the development of the PathFinder algorithm—a perfect example of "necessity being the mother of invention." As part of an FPGA architecture exploration tool called Emerald [11], PathFinder was also employed in the development of an FPGA under development by IBM in the mid-1990s. This was particularly appropriate because PathFinder is inherently architecture independent. That experience showed that PathFinder was indeed an improvement over other FPGA routers available at the time.

The PathFinder algorithm was adopted and carefully implemented by Betz and Rose in the very popular versatile place and route (VPR) FPGA tool suite [12, 13], which has been widely used for academic and industry research. The Toronto place-and-route challenge [14] was established as a way to compare different FPGA placement and routing algorithms. Since the contest was established in 1997, the champion has been either VPR's implementation of PathFinder or SC-PathFinder, implemented at the University of California–Santa Cruz. Although companies are reluctant to divulge the details of their design tools, it is clear that some version of the PathFinder algorithm is currently used by virtually all commercial FPGA routers.

17.2 THE PATHFINDER ALGORITHM

17.2.1 The Circuit Graph Model

One of the key features of PathFinder is its architecture independence, which derives from the use of a simple underlying graph representation of FPGA architectures. This model allows PathFinder to be adapted to virtually any architecture and thus used to explore new architectures with very little startup cost. Once an architecture has been decided on, PathFinder can be specialized to it for improved results and performance.

The routing resources in an FPGA and their connections are represented by the directed graph $G = (V, E)$. The set of vertices V corresponds to the electrical nodes or wires in the FPGA architecture, and the edges E correspond to the switches that connect these nodes. An example of this graph model is shown in Figure 17.1 for a version of the Triptych FPGA cell. Note that devices are represented only implicitly by the wires connected to their terminals. That is, routing from one device terminal to another is routing between the wires connected to those terminals.

Associated with each node n in the architecture is a base cost b_n that represents the relative cost of using that node. This cost is typically proportional to the length of the wire, although other measures like capacitance or number of fanins and fanouts are also possible. Each node also has a delay d_n, which may or may not be the same as b_n.

Given a signal i in a circuit mapped onto the FPGA, the signal net N_i is the set of terminals, including the source terminal s_i and sinks t_{ij}. N_i forms a subset of V. A solution to the routing problem for signal i is the directed routing tree RT_i embedded in G and connecting the source s_i to all of its sinks t_{ij}.

17.2.2 A Negotiated Congestion Router

We assume that the reader is familiar with Djikstra's shortest-path graph algorithm [15–17], which is at the core of many routing algorithms. Note that in our formulation costs are associated with nodes, not edges. This changes the basic

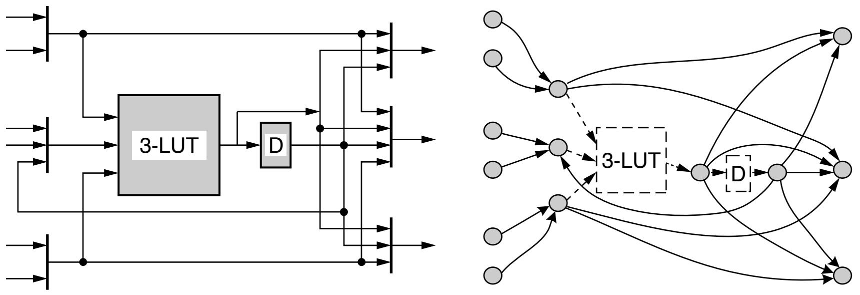

FIGURE 17.1 ■ The circuit for a Triptych FPGA cell is represented in PathFinder by the graph at the right.

shortest-path algorithm only slightly by redefining the cost of a path from node n_i to node n_j as the sum of the node costs along the path, including the starting and ending nodes.

Routing algorithms differ primarily in the cost function applied to the routing resources and in how individual applications of the shortest-path algorithm are used to successfully route all the signals of a netlist onto the graph representing the architecture. We ignore the issue of fanout in our initial presentation and assume that each signal is a simple route from source to a single sink.

A naive routing algorithm proceeds by applying the shortest-path algorithm to each signal in order, with the cost of a node defined as

$$c_n = b_n \tag{17.1}$$

Resources already used by previous routes are not available to later routes. It is clear that the order in which signals are routed is crucial, as later routes have many fewer available routing resources. Some algorithms perform rip-up and retry when later routes cannot find a path. Selected early routes that are blocking are ripped up and rerouted later—in essence, adaptively changing the order in which signals are routed.

The very simple example in Figure 17.2 shows how this naive algorithm can fail. There are three signals, 1, 2, and 3, to be routed from the sources S_1, S_2, and S_3 to their respective sinks D_1, D_2, and D_3. The ovals represent partial paths through one or more nodes, annotated with the associated costs. Ignoring congestion, the minimum-cost path for each signal would use node B. If the naive obstacle avoidance routing scheme is used, the order in which the signals are routed becomes crucial: Routing in the order 1, 2, 3 fails, and the minimum-cost routing solution will be found only when starting with signal 2.

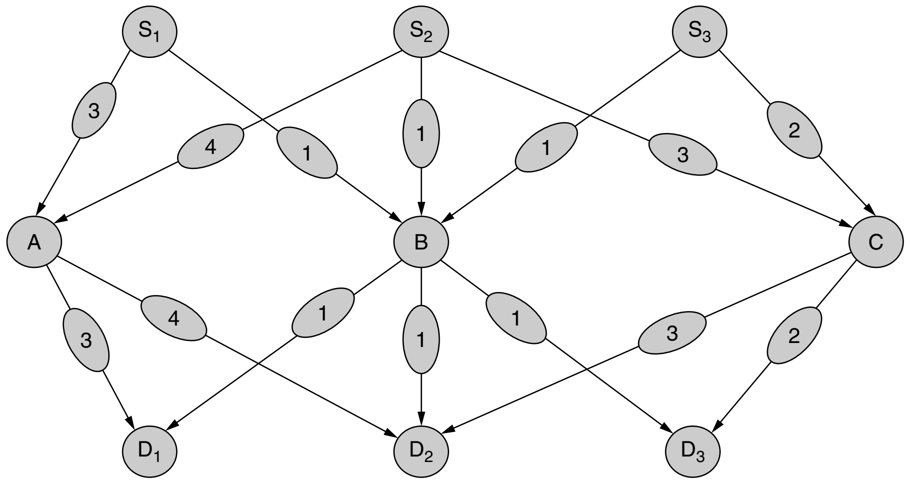

FIGURE 17.2 ■ First-order congestion.

This problem can be solved by introducing negotiated congestion avoidance, first suggested by Nair [18] by extending the cost of using a given node n in a route to

$$c_n = b_n \cdot p_n \tag{17.2}$$

where b_n is the base cost of using n, and p_n is a function of the number of other signals presently using n (p_n is often called the "present-sharing" term). Note that in the naive router, $p_n = 1$ if no other signals are using n, and infinity otherwise. In the negotiated congestion algorithm, p_n is set initially to 1 and all signals are routed. This allows each signal to be routed as if no other signals were present. The cost of sharing is then increased, and all nets are ripped up and rerouted in turn. This iterative process continues, with the cost of sharing increasing at each iteration until all signals have been successfully routed. The idea is that the cost of a congested node will increase and that signals that have other alternatives will eventually find other paths, leaving the node to the signal that needs it most. p_n is a function of the iteration i and the number of signals sharing a node k. The definition of p_n is a key tuning parameter of PathFinder.

The negotiated congestion avoidance algorithm solves the problem of Figure 17.2. During the first iteration, p_n is initialized to 1, and consequently no penalty is imposed for the use of n regardless of how many signals occupy it. Thus, in the first iteration all three signals share B. When the sharing function p_n increases sufficiently, signal 1 will find that a route through node A gives a lower cost than a route through the congested node B. During an even later iteration signal 3 will find that a route through node C gives a lower cost than that through B. This scheme of negotiation for routing resources depends on a relatively gradual increase in the cost of sharing nodes. If the increase is too abrupt, signals may be forced to take high-cost routes that lead to other congestion. Just as in the standard rip-up and retry scheme, the ordering becomes important.

While iterative negotiated congestion routing with the cost function of equation 17.2 can optimally route simple "first-order" routing problems like that in Figure 17.2, it fails on more complex "second-order" routing problems like that shown in Figure 17.3. Again we need to route three signals, one from each source to the corresponding sink. Let us first consider this example from the standpoint of obstacle avoidance with rip-up and retry. Assume that we start with the routing order (1, 2, 3). Signal 1 routes through node B, and signals 2 and 3 share node C. For rip-up and retry to succeed, both signals 1 and 2 would have to be rerouted, with signal 2 rerouted first. Because signal 1 does not use a congested node, determining that it needs to be rerouted is in general difficult.

This second-order congestion problem cannot be solved using p_n alone. Signal 2 will never choose node B because the present sharing costs for nodes B and C are the same, with B used by signal 1 and C used by signal 3. Since the path through C is cheaper, it is always chosen. PathFinder solves this by extending the cost function with a "history" term, h_n:

$$c_n = (b_n + h_n) \cdot p_n \tag{17.3}$$

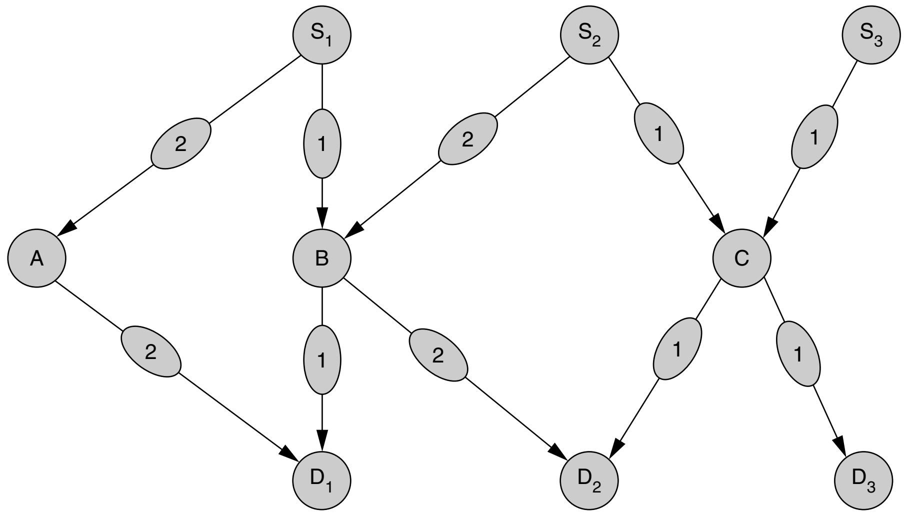

FIGURE 17.3 ▪ Second-order congestion.

Unlike p_n, h_n "remembers" the congestion that has occurred on node n during previous routing iterations. That is, the history term is updated after each routing iteration; any node shared by multiple signals has its history term increased by some amount. The effect of h_n is to permanently increase the cost of using congested nodes so that routes through other nodes are attempted. Without this term, as soon as signals stop sharing a node, its cost drops to the base cost and it again becomes attractive. This leads to oscillations where signals switch back and forth between nodes but never resolve the congestion problem. The addition of the history term is a key difference between PathFinder and Nair's routing algorithm [18].

The term h_n allows the problem in Figure 17.3 to be routed successfully. On each iteration that node C is shared, h_n is increased slightly. When signal 2 switches to using node B, the cost of node C remains elevated. Now the history cost of node B rises because it is shared by signals 1 and 2. Eventually signal 1 will route through node A. Note that, depending on the base costs and how p_n and h_n are defined, signal 2 may switch back and forth between nodes B and C several times before the history costs of both are sufficiently high to force signal 1 onto node A.

The history term h_n is updated whenever a node n has shared signals. The size of δ_h, the amount by which h_n is increased, and how this depends on k, the number of sharing signals, are tunable parameters. If δ_h is too small, many iterations may be required to resolve the congestion; if it is too large, some solutions may not be found. Additionally, the relationship between p_n and h_n is very important. For example, it can be important to give the history term a chance to solve congestion before forcing the issue with p_n.

The details of the Negotiated Congestion algorithm are given in Figure 17.4. The while loop at line 2 executes the routing iterations until a solution has been

```
iteration $\leftarrow$ 0                                                1
While shared resources exist                                            2
    Iteration $\leftarrow$ iteration + 1                                3
    Loop over all signals $i$ (signal router)                           4
        Rip up routing tree $RT_i$                                      5
        $RT_i \leftarrow s_i$                                           6
        Loop until all sinks $t_{ij}$ have been found                   7
            Initialize priority queue $PQ$ to $RT_i$ at cost 0          8
            Loop until new $t_{ij}$ is found                            9
                Remove lowest cost node $m$ from $PQ$                   10
                Loop over fanouts $n$ of node $m$                       11
                    Add $n$ to $PQ$ at cost $P_{im} + c_n$              12
                end loop                                                13
            end loop                                                    14
            Loop over nodes $n$ in path $t_{ij}$ to $s_i$ (backtrace)   15
                Update $c_n$                                            16
                Add $n$ to $RT_i$                                       17
            end loop                                                    18
        end loop                                                        19
    end loop                                                            20
    Loop over all nodes $n_i$ shared by multiple signals                21
        $h_i \leftarrow h_i + \delta(k)$                                22
    end loop                                                            23
end while                                                               24
```

FIGURE 17.4 ■ Negotiated Congestion algorithm.

found. The signal router loop at line 4 iterates over all signals in the netlist, ripping up and rerouting the nets one at a time. The routing tree RT_i is the set of nodes used to route signal i. To reroute a signal, the routing tree is reset to be just the signal's source.

The priority queue is used to implement the breadth-first search of Djikstra's algorithm. At each iteration of the loop of line 9, the lowest-cost node is taken from the priority queue. It is generally best to order the nodes with the same cost according to when they were inserted into the queue, with the newest nodes being extracted first. The cost used when inserting a new node in the priority queue at line 12 is

$$P_{im} + c_n \tag{17.4}$$

where P_{im} is the cost of the current partial path from the source, and c_n is the cost of using node n.

A signal is routed one sink at a time using Djikstra's breadth-first algorithm. When the search finds a sink, the nodes on the path from the source to it are added to RT_i. This is done by back-tracing the search path to the source. The search is then restarted with the priority queue being initialized with all the nodes already in RT_i. In this way, all the nodes on routes to previously found sinks are used as potential sources for routes to subsequent sinks. This algorithm for constructing the routing tree is similar to Prim's algorithm for determining a minimum spanning tree over an undirected graph, and it is identical to one suggested by Takahashi and Matsuyama [19] for constructing a tree embedded in an undirected graph. The quality of the points chosen by the algorithm is an open question for directed graphs; however, finding optimum (or even near-optimum) points is not essential for the router to be successful in adjusting costs to eliminate congestion.

The VPR router [12] reduces the cost of reinitializing the priority queue for each fanout by observing that for large-fanout nets, most of the paths found in searching for the previous fanout remain valid, especially if the segment added to the routing tree is relatively small. Thus, the search continues from the previous state after the new segment has been added to the routing tree. Because of the way Djikstra's algorithm ignores nodes after they have been visited once, this optimization must be implemented carefully to avoid expensive routing trees for high-fanout nets. Other algorithms for forming the fanout tree are possible. For example, there are times when routing to the most distant sink first results in a better routing tree.

At the end of each iteration, the history cost of each node shared by multiple signals is updated. The δ added to the history cost is generally a function of k, the number of signals sharing the node.

17.2.3 The Negotiated Congestion/Delay Router

To introduce delay into the Negotiated Congestion algorithm, we redefine the cost of using node n when routing a signal from s_i to t_{ij} as

$$C_n = A_{ij} d_n + (1 - A_{ij}) c_n \tag{17.5}$$

where c_n is defined in equation 17.3 and A_{ij} is the slack ratio:

$$A_{ij} = D_{ij}/D_{\max} \tag{17.6}$$

where D_{ij} is the delay of the longest delay (register–register) path containing the signal segment (s_i, t_{ij}), and $D_{\max}$ is the maximum delay over all paths (i.e., the critical-path delay). Thus, $0 < A_{ij} \leq 1$. (This standard definition of slack ratio is easily extended to include circuit inputs and outputs with timing constraints as well as circuits with multiple clocks.)

Because path delay is made up of both device and wire delay, and the router can only control the wire delay, a more accurate formulation for A_{ij} is

$$A_{ij} = (D_{ij} - Ddev_{ij})/(D_{\max} - Ddev_{ij}) \tag{17.7}$$

where $Ddev_{ij}$ is the path delay from node i to node j attributable to devices, and $D_{ij} - Ddev_{ij}$ is thus the wire delay on the path from node i to node j. With equation 17.7, paths with the same path delay but greater wire delay pay more attention to delay and less to congestion.

The first term of equation 17.5 is the delay-sensitive term; the second term is congestion sensitive. Equations 17.5, 17.6, and 17.7 are the keys to providing the appropriate mix of minimum-cost and minimum-delay trees. If a particular source/sink pair lies on the critical-path, then $A_{ij} = 1$ and the cost of node n

is just the delay term; hence a minimum-delay route is used and congestion is ignored. In practice, A_{ij} is limited to a maximum value such as 0.9 or 0.95 so that congestion is not completely ignored. If a source/sink pair belongs to a path whose delay is much smaller than the critical-path, then A_{ij} is small and the congestion term dominates, resulting in a route that avoids congestion at the expense of extra delay.

To accommodate delay, the basic Negotiated Congestion algorithm of Figure 17.4 is changed as follows. For the first iteration, all A_{ij} are initialized to 1 and minimum-delay routes are found for every signal. This yields the smallest possible critical-path delay. All A_{ij} are recomputed after every routing iteration using the critical-path delay and the delays incurred by signals on that iteration.

The sinks of each signal are now routed in decreasing A_{ij} order. This allows the most timing-constrained sinks to determine the coarse structure of the routing tree with no interference from less constrained paths.

The priority queue (line 8 in Figure 17.4) is initialized by inserting each node of RT_i with the cost $A_{ij} \sum_k d_k$, where the n_k are nodes on the path from the source n_i to node n_j. This initializes the nodes already in the partial routing tree with the weighted path delay from the source.

The router completes when no more shared resources exist. Note that by recalculating all A_{ij}, we have kept a tight rein on the critical-path. Over the course of the routing iterations, the critical-path increases only to the extent required to resolve congestion. This approach is fundamentally different from other schemes [4, 5] that attempt to resolve congestion first and then reduce delay by rerouting critical nets.

The PathFinder algorithm is particularly powerful for asymmetric architectures that have a range of slow and fast wires. By making the slower wires lower cost, the negotiation algorithm automatically assigns critical signals to the fast wires as needed and noncritical signals to the slow wires.

17.2.4 Applying A* to PathFinder

Djikstra's shortest-path algorithm performs an expensive breadth-first search of the graph. This search has an $O(n^2)$ running time for two-dimensional circuit structures, where n is the length of the path. The A* heuristic [20] is a technique that uses additional information about the cost of paths in the graph to bound the size of the search. The cost of a partial path becomes the cost of the partial path plus the estimated cost from the end of the partial path to the destination. If this estimated cost is a lower bound on the actual cost, then the search will provide an optimal solution. If the estimated cost is accurate, then the search becomes a depth-first search with $O(n)$ running time.

In applying A* to PathFinder, both the cost and the delay of paths in the graph must be estimated. We modify equation 17.4 as follows:

$$C_n = P_{im} + A_{ij}(d_n + Dest_{nj}) + (1 - A_{ij})(c_n + Cest_{nj}) \qquad (17.8)$$

where $Dest_{nj}$ and $Cest_{nj}$ are the estimated delay and cost, respectively, of the minimum-delay route from n to sink j.

To use the A* heuristic, the router must know the destination in order to determine the estimated cost. Instead of letting the breadth-first router find the closest destination when there are multiple fanouts, the path length estimates are used to sort the fanouts from closest to furthest and the routing is performed in this order.

In many FPGAs, such as those that are standard island style, the cost and delay of routes can be estimated based on the locations of the source and destination using the geometry of the layout. A more general and accurate method is to use the shortest-path algorithm to create a complete “distance table” that contains the cost estimate of the minimum-delay route from every node to all potential sinks. This is only feasible, however, for relatively small architectures or for coarse-grained architectures that have many fewer nodes than fine-grained FPGAs. To reduce the table size, clustering can be used and estimates stored for the cost/delay between clusters [21]. If the cost/delay between two clusters is taken as the minimum cost/delay between any two nodes in the two clusters, it represents a true lower bound. Clustering has been reported to reduce the size of the distance table by a factor of 100 while slowing the search only by a factor of 2 [21].

In the early iterations of PathFinder, when sharing is ignored, the full advantage of A* is obtained. That is, if the cost/delay estimates are accurate, a depth-first search is achieved. As the cost of sharing rises, however, the cost estimates, which do not include the sharing costs, become less and less accurate and the search becomes less efficient.

In experiments with PathFinder and A*, Swartz et al. [22] used a multiplicative direction factor α to inflate the path estimate. In effect, α determines how aggressively the router drives toward the target sink. An α of 1.0 corresponds to true A* and is guaranteed to find the shortest source/sink connection. Swartz et al. determined that an α of 1.5 gave the best results for large circuits, with no measurable degradation in the quality of the resulting routing. However, note that the cost function had only a congestion term and no delay term. Tessier also experimented with accelerating routing with even more aggressive use of the A* search [23, 24].

17.3 ENHANCEMENTS AND EXTENSIONS TO PATHFINDER

Many research papers have discussed extensions and optimizations of the PathFinder algorithm. First and foremost is the work by Betz and Rose on VPR [12], which for the past eight years has been a widely used vehicle for academic and industrial research into FPGA architectures and CAD. We discuss here some of the more salient ideas that have been applied to PathFinder.

17.3.1 Incremental Rerouting

A common optimization suggested in the original PathFinder paper [8] is to limit the rip-up and rerouting of signals in an iteration only to those that use shared resources. Intuitively, this reduces the amount of “wasted” effort that goes into rerouting signals that always take the same path. The argument is that if a signal does not use a shared resource, it will take the same path as it did before, because history costs can only rise and thus no other path can become cheaper. This argument fails where p_n becomes smaller as sharing signals reroute around a congested node. Experience shows that this optimization increases the number of routing iterations, but reduces the total running time substantially, with negligible impact on the quality of the solution found.

17.3.2 The Cost Function

There are many ways to tune PathFinder for specific architectures or to achieve specific goals. Many variations of the cost function have been described that change how the three cost terms b_n, p_n, and h_n are computed and combined. The essential feature of the cost function is that h_n is a function of the history of the congestion of the node and that p_n is a function of the current congestion. The rates at which h_n and p_n increase can be tuned; increasing them quickly, for example, decreases the number of iterations required but also decreases the quality of the solution. The history term may include a decay function on the assumption that the more recent history is more valid than the distant past. This is particularly important when PathFinder is used in an integrated place-and-route tool [21, 25].

The PathFinder cost function can also be modified to include both short-path and long-path delay terms [26]. For long paths, delay is minimized by using the PathFinder cost function. For short paths, however, the cost function is changed to find a path with a target delay, not the minimum delay. This changes the underlying shortest-path problem considerably and requires an accurate "look-ahead" function that predicts the remaining delay to the destination so that the router can opportunistically add the appropriate extra delay.

17.3.3 Resource Cost

Determining the base cost of routing resources is harder than it appears. The shortest-path algorithm attempts to minimize the total cost of a solution, so minimizing the cost should also minimize congestion. The typical cost function used by routers is the length of the wire, which is a good heuristic for typical architectures where the number of available wires is inversely proportional to their individual lengths. A better heuristic is to base the cost of a wire on the expected routing demand for it. This can be approximated by routing a set of placed benchmarks onto an architecture and measuring wire by wire the routing demand. Another method is to perform a large number of random routes using a typical Rent's wirelength distribution through the architecture and again measuring the overall use of each wire. In this formulation, wire costs are initialized to 1, raised à la PathFinder according to wire usage, and converge to some constant value.

Delay is an approximation that is often used for cost as it is typically closely related to wirelength and relative demand. It also simplifies the cost function for the integrated congestion and delay router.

17.3.4 The Relationship of PathFinder to Lagrangian Relaxation

The PathFinder algorithm is very similar to Lagrangian relaxation for finding an optimal routing subject to congestion and delay constraints [27–29]. In Lagrangian relaxation, the constraints are relaxed by multiplying them by a vector of Lagrangian multipliers and adding them to the objective function to be minimized. The solution to a Lagrangian formulation with a specific set of Lagrangian multipliers provides an approximate solution to the original minimization problem. An iterative procedure that modifies the Lagrangian multipliers is used to find increasingly better solutions. A subgradient method is used to update the multipliers. Intuitively, the multipliers are increased or decreased depending on the extent to which the corresponding constraint is satisfied.

A Lagrangian relaxation method proceeds somewhat differently from the PathFinder algorithm. The multipliers operate much like PathFinder's history term, but there is no corresponding present-sharing term p_n. While the history term is monotonically nondecreasing, the Lagrangian multipliers can both increase and decrease depending on how well the corresponding constraint is satisfied. The amount by which the multipliers are adjusted in Lagrangian relaxation is also decreased with each iteration.

17.3.5 Circuit Graph Extensions

The simple circuit graph model is very general, but there are some specific circuit structures that require extensions. This section describes some solutions for these.

Symmetric device inputs

Lookup tables (LUTs) are the prime example of FPGA devices whose pins are "permutable." That is, the inputs to a LUT can be swapped arbitrarily by permuting the table's contents. Other devices like adders also have symmetric inputs. In the simple graph model, a signal is routed to a specific input terminal and there is no way to specify a route to one of a set of terminals.

Symmetric inputs are easily accommodated in the graph model by adding "pseudo-multiplexers" on the inputs of the LUT. These are shown as dashed nodes at the top of Figure 17.5. Signal sinks can be arbitrarily assigned to the LUT inputs and routed in the usual way. After the routing solution has been found, the pseudo-multiplexers are removed and implemented "virtually" by permuting the LUT table contents appropriately. In the example of Figure 17.5, the signals a, b, and c are routed to the LUT inputs A, B, and C, respectively, using the pseudo-multiplexers as shown with bold lines. This routing is then used to permute the LUT inputs as shown on the right by modifying the LUT contents.

De-multiplexers

A de-multiplexer is a device that can connect its input to at most one of several outputs. Each output connection is represented as an edge in the circuit graph shown in Figure 17.6. Wire fanout, of course, is not constrained, and there is no way in the graph model to specify a constraint on the number of fanouts that can be used. This case is handled by a special counter that counts the number of the edges that are used. If more than one edge is being used, the

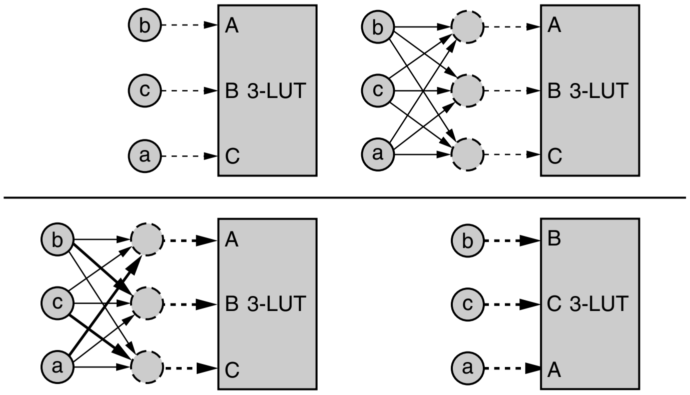

FIGURE 17.5 ■ Symmetric device inputs are handled by inserting pseudo-multiplexers.

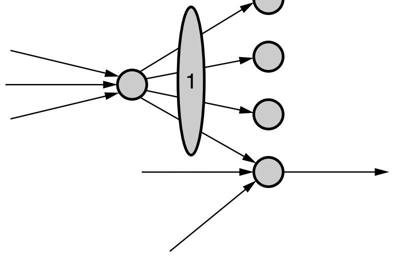

FIGURE 17.6 ■ De-multiplexers are handled by negotiating for the fanouts of the de-multiplexer.

de-multiplexer is being shared in much the same way that wires can be shared by signals. A PathFinder cost function can be applied with both a sharing and a history component so that the single fanout used is determined by means of negotiation.

Bidirectional switches

Edges in the graph model, which represent connections, are directional. This models multiplexer-based architectures directly. Transistors that are often used to construct configurable interconnects are bidirectional. These bidirectional switches simply translate to two directional edges in the graph. The router uses at most one of the edges, which induces a logical direction on the switch. That is, when a switch is turned on in a configuration, it is being driven by an output from one side to the other.

17.4 PARALLEL PATHFINDER

A typical large FPGA design has many thousands of signals. If separate signals could be routed in parallel, the degree of parallelism would be limited only by the number of signals to be routed and the number of processors available. The difficulty, of course, is that the route taken by each signal depends on the knowledge of other signal routes, as routing resources cannot be shared. Although parallel implementations of global standard cell routers exist, the problem for FPGAs becomes much harder because the routing resources are discrete and fixed.

Because the routing of separate signals in an FPGA is tightly coupled, it might appear that a parallel approach to routing FPGAs would not be possible given that knowledge of other signal locations is necessary to find a feasible route. This is the case in a typical maze router, which uses rip-up and reroute to resolve conflicts. In PathFinder, however, there is no restriction on the number of signals that can occupy a resource simultaneously during routing. Instead, the cost of using congested resources is the mechanism used to resolve resource conflicts. If the congestion costs are decentralized in a parallel environment, the concerns are how and when they will be updated and whether the update method will be acceptable in terms of the number of processors effectively utilized and the quality of the resulting routing.

In Chan et al. [30] a distributed memory multiprocessor implementation of the PathFinder algorithm is described. Each processor has a private local memory and is connected in a network. Processors communicate with each other by sending and receiving messages via Unix socket communication. A complete copy of the routing resource graph, including first- and second-order congestion costs, is kept and maintained by each processor. The signals in a netlist to be routed are statically assigned to processors such that each processor has about the same number of sinks to be routed. No attempt is made to assign signals to processors based on locality.

Processors route signals asynchronously and thus communicate updated congestion costs asynchronously. There is no guarantee of the order or the timing of the arrival of such congestion cost updates, resulting in a source of indeterminism. Processors are allowed to proceed to successive iterations without waiting for others, although a limit of a few iterations of separation is generally employed.

It is conceded that, because of latency, this parallel routing algorithm may not converge. Imagine a scenario in which two signals being routed by two different processors vie for the same resource. Message latency or merely concurrency may cause the two signals to oscillate between routing iterations, because each processor knows where the other processor's signal was in the last iteration but not in the current one. Such cases generally occur during the last iterations of a route. At that point, Chan and colleagues [30] reduce the multiprocessor implementation to a single-processor implementation in order to resolve the congestion.

This parallel implementation was tested on a set of benchmarks ranging from 118 to 1542 signal nets on the Xilinx 4000 architecture. Speedups ranged from 1.6 to 2.2 times for two processors and 2.3 to 3.8 times for four processors. For nearly all benchmarks, no additional speedups are obtained for more than four processors. The performance of the benchmarks (in terms of delay or clock rate) was shown to vary minimally with increasing numbers of processors.

This initial implementation of a parallel form of PathFinder is significant in that it demonstrates appreciable speedups while employing a rather simple computational framework. Because of the inherent approximations of congestion cost and its gradual increase, PathFinder exhibits good qualities for parallelism in a framework where congestion costs are communicated asynchronously, as they become available. It may result (as shown by Chan et al. [30]) in an increased number of iterations to converge, but is able to employ more multiple loosely connected processors to good advantage.

17.5 OTHER APPLICATIONS OF THE PATHFINDER ALGORITHM

PathFinder has been used to incrementally reroute signals around faults in cluster-based FPGAs [31]. This rerouting uses the accumulated history costs acquired by the initial routing to quickly find a new routing solution when nodes and edges in the circuit graph have been removed because of faults.

QuickRoute [32] extends PathFinder to handle pipelined routing structures. The key idea in QuickRoute is to change Djikstra's shortest-path algorithm to allow nodes to be visited more than once, by paths with different latencies. This causes many more overlapping paths to be explored, but the negotiated congestion avoidance of PathFinder still performs well.

Several groups have applied PathFinder to the problem of scheduling the communication in computing graphs to coarse-grained architectures or multiprocessors [33–35]. In this application of PathFinder, the routing becomes a space–time problem.

17.6 SUMMARY

The widespread use of PathFinder by commercial FPGA routers and university research efforts alike is a testimonial to its robustness.

Several key facets of the algorithm make it attractive. However, its primary advantage is the iterative nature of resolving congestion, using both current as well as historical resource use in the formulation of the cost function. By very gradually increasing cost due to both usages, the routing search space is thoroughly explored. Routing with other objective functions, delay in particular, is easily integrated into the cost function. A primary feature implicit in PathFinder (that distinguishes it from previous efforts) is the allowance of nonphysically feasible intermediate states—for example, shared resources—while converging to a physically feasible final state. Finally, by being grounded in a directed graph representation, PathFinder is very adaptable to changing FPGA architectures as well as other problems that can be abstracted to a directed graph.

In the future we see the routing problem as being an increasingly dominant hurdle in the use of FPGAs with millions of resources. To reduce the runtime, more investigation will be required to effectively parallelize PathFinder, making use of additional computational resources. Given the growing focus on other objectives such as power consumption, it is likely that we will see experimentation with other cost function formulations as well.

Acknowledgments We wish to thank Gaetano Borriello for initial discussions about routing when PathFinder was being applied to the Triptych architecture, and Steven Yee for his help in constructing detailed descriptions of the Xilinx architectures. We also thank Pak Chan and Martine Schlag for sharing the results on parallel PathFinder.

References

[1] W. A. Dees, R. J. Smith. Performance of interconnection rip-up and reroute strategies. *Design Automation Conference*, 1981.

[2] R. Linsker. An iterative-improvement penalty-function-driven wire routing system. *IBM J. Res. Development* 28(5), 1984.

[3] J. Cohn, D. Garrod, R. Rutenbar, L. Carley. Koan/anagram II: New tools for device-level analog placement and routing. *IEEE Journal of Solid-State Circuits* 26(3), 1991.

[4] S. Brown, J. Rose, Z. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11(5), 1992.

[5] J. Frankle. Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. *Design Automation Conference*, 1992.

[6] M. J. Alexander, J. P. Cohoon, J. L. Ganley, G. Robins. An architecture-independent approach to FPGA routing based on multi-weighted graphs. *Proceedings of the Conference on European Design Automation*, 1994.

[7] M. Palczewski. Plane parallel a maze router and its application to FPGAs. *Design Automation Conference*, 1992.

[8] L. McMurchie, C. Ebeling. A negotiation-based performance-driven router for FPGAs. *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays Aided Design,* 1995.

[9] G. Borriello, C. Ebeling, S. Hauck, S. Burns. The triptych FPGA architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3(4), 1995.

[10] C. Ebeling, L. McMurchie, S. Hauck, S. Burns. Placement and routing tools for the triptych FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3(4), 1995.

[11] D. C. Cronquist, L. McMurchie. Emerald: An architecture-driven tool compiler for FPGAs. *Proceedings of the Fourth ACM International Symposium on Field-Programmable Gate Arrays*, 1996.

[12] V. Betz, J. Rose. VPR: A new packing, placement and routing tool for FPGA research. *Proceedings of the Seventh International Workshop on Field-Programmable Logic and Applications*. Springer-Verlag, 1997.

[13] V. Betz, J. Rose, A. Marquardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic, 1999.

[14] V. Betz. The FPGA place-and-route challenge (*www.eecg.toronto.edu/vaughn/challenge/challenge.html*).

[15] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik* 1(1), December 1959.

[16] E. Moore. The shortest path through a maze. *International Symposium on the Theory of Switching*, April 1959.
[17] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers* 10, September 1961.
[18] R. Nair. A simple yet effective technique for global wiring. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6(2), 1987.
[19] H. Takahashi, A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Japonica* 24(6), 1980.
[20] P. Hart, N. Nilsson, B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 1968.
[21] A. Sharma. *Place and Route Techniques for FPGA Architecture Advancement*, Ph.D. thesis, University of Washington, 2005.
[22] J. S. Swartz, V. Betz, J. Rose. A fast routability-driven router for FPGAs. *Proceedings of the ACM/SIGDA Ssixth International Symposium on Field-Programmable Gate Arrays*, 1998.
[23] R. G. Tessier. Negotiated A* routing for FPGAs. *Fifth Canadian Workshop on Field-Programmable Logic*, 1998.
[24] R. G. Tessier. *Fast Place and Route Approaches for FPGAs*, Ph.D. thesis, MIT, 1999.
[25] A. Sharma, S. Hauck, C. Ebeling. Architecture-adaptive routability-driven placement for FPGAs. *International Conference on Field-Programmable Logic and Applications*, 2005.
[26] R. Fung, V. Betz, W. Chow. Simultaneous short-path and long-path timing optimization for FPGAs. *IEEE/ACM International Conference on Computer Aided Design*, 2004.
[27] S. Lee, Y. Cheon, M. D. F. Wong. A min-cost flow based detailed router for FPGAs. *International Conference on Computer-Aided Design*, 2003.
[28] S. Lee, M. Wong. Timing-driven routing for FPGAs based on Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22(4), 2003.
[29] M. M. Ozdal, M. D. F. Wong. Simultaneous escape routing and layer assignment for dense PCBs. *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design,* 2004.
[30] P. K. Chan, M. D. F. Schlag, C. Ebeling, L. McMurchie. Distributed-memory parallel routing for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19(8), August 2000.
[31] V. Lakamraju, R. Tessier. Tolerating operational faults in cluster-based FPGAs. *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field-Programmable Gate Arrays*, 2000.
[32] S. Li, C. Ebeling. QuickRoute: A fast routing algorithm for pipelined architectures. *IEEE International Conference on Field-Programmable Technology*, 2004.
[33] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Design, Automation and Test in Europe*, 2003.
[34] J. Cook, L. Baugh, D. Gottlieb, N. Carter. Mapping computation kernels to clustered programmable reconfigurable processors. *IEEE International Conference on Field-Programmable Technology,* 2003.
[35] L.-Y. Lin, C.-Y. Wang, P.-J. Huang, C.-C. Chou, J.-Y. Jou. Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. *Proceedings of the 2005 Conference on Asia South Pacific Design Automation*, 2005.