perature equals the maximum cost change caused by any one move at that temperature.

The schedule of Lam and Delosme [83], employs feedback control to set the annealing schedule. It monitors the standard deviation of the cost, the average cost, and the fraction of proposed moves that were accepted, α, over the past τ moves. Typically τ is 100. These values are inputs to a sophisticated feedback system that determines a new temperature. In this schedule, a new temperature is computed every move — that is, the "inner loop" in Figure 2.9 executes only one iteration each time control reaches it. The anneal terminates when there has been no change in the average cost for the last $k \cdot \tau$ moves, where k is typically 5. This annealing schedule also employs a range limiter to control the move generation process. The R_{limit} parameter in Figure 2.9 controls how close together blocks must be to be considered for swapping. Initially, R_{limit} is fairly large, and swaps of blocks far apart on a chip are likely. Throughout the anneal, R_{limit} is adjusted to try to keep the fraction of moves accepted at any temperature close to 0.44. If the fraction of moves accepted, α, is less than 0.44, R_{limit} is reduced, while if α is greater than 0.44, R_{limit} is increased.

One disadvantage of the Lam schedule is its complexity. Fortunately, Swartz and Sechen have developed an annealing schedule incorporating some of the key ideas of the Lam schedule, and which achieves equivalent quality, but is much less complex [84]. In this schedule, the number of moves attempted in the "inner loop" is $10 \cdot N_{blocks}^{1.33}$. The range limiter, R_{limit} is updated according to a fixed (hard-coded) schedule; the exact form of R_{limit} is not specified in [84], but it likely initially spans the entire chip, and gradually shrinks to a small region. The "outer loop" is executed 150 times, and then the anneal terminates. The temperature is controlled by the fraction of moves accepted:

$$T_{new} = \left[1 - \frac{\alpha - 0.44}{40}\right] \cdot T_{old}, \qquad (2.3)$$

where 0.44 is desired acceptance rate, and 40 is a damping coefficient to prevent wild oscillations in temperature. While this schedule is much simpler than Lam's, it has sacrificed some of the adaptability of the Lam schedule, since the range limiter variation and the number of outer loop iterations are now hard-coded.

Since the amount of routing in FPGAs is limited and set by the manufacturer when the FPGA is fabricated, some FPGA placement tools attempt to optimize not just the wirelength of a placement, but also its routability. In [85], Ebeling et al describe a simulated annealing based placer that targets the Triptych FPGA developed at the University of Washington. Its cost function incorporates not only a bounding-box wirelength term, but also a "porosity" term that monitors the fraction of logic blocks

in a local area that are being used. Since the Triptych FPGA is usually unroutable in regions where the logic blocks are completely used, maintaining a porosity of 50% or so across the FPGA is essential. In [86, 87], Nag and Rutenbar describe a simulated annealing based tool that performs placement and routing simultaneously in one combined step. After any swap of blocks, all the affected nets are re-routed via a maze router. To keep the CPU time reasonable, this maze router is constrained to look at only a small number of potential routes when the temperature is high; at lower temperatures, the router is allowed to spend more time looking for routes. If no suitable route is found among the allowed candidates, the net is marked as currently unroutable, and the placement cost is increased. The result quality of this tool is high, but the CPU time required is very large — a circuit containing only 461 Xilinx 4000 logic blocks required 11 hours of CPU time to place and route, and the complexity of this algorithm appears to be approximately $O(n^3)$, where n is the number of logic blocks in a circuit. Alexander et al have created a partitioning-based algorithm, FPR [88], that performs placement and global routing simultaneously, again in an attempt to maximize circuit routability.

Some work has also been done in routability-driven standard cell placement that is applicable to FPGAs. Cheng [89] describes a simulated annealing placer that divides a chip into many subregions and estimates the demand for wiring in each region. This demand is compared to the supply of routing in each region, and when the expected demand outstrips the routing supply in some regions the placement is penalized by having its cost increased. Dividing the chip into subregions and estimating wiring demand makes localized congestion visible, and merely estimating the wiring demand in a region is faster than actually routing each placement proposed during the anneal.

2.2.3 Routing

Once locations for all the logic blocks in a circuit have been chosen, a router determines which programmable switches should be turned on to connect all the logic block input and output pins required by the circuit. In FPGA routing, one usually represents the routing architecture of the FPGA as a directed graph [86, 85]. Each wire and each logic block pin becomes a node in this *routing-resource graph* and potential connections become edges. Some prior research has represented FPGAs as undirected graphs [90], but a directed graph representation is needed if directional switches, like tri-state buffers and multiplexers, are to be modelled correctly.

Routing a connection corresponds to finding a path in this routing-resource graph between the nodes representing the logic block pins to be connected. To avoid using up too many of the limited number of wires in an FPGA, one wants this path to be as short as possible. As well, it is important that the routing for one net not use up rout-

ing resources another net needs, so most FPGA routers have some kind of congestion avoidance scheme to resolve contention for routing resources. An additional optimization goal is to make nets on or near the critical path fast by routing them using short paths and fast routing resources. Routers that attempt to optimize timing in this way are called timing-driven, whereas delay-oblivious routers are purely routability-driven. Since most of the delay in FPGAs is due to the programmable routing, timing-driven routing is crucial to obtain good circuit speeds.

FPGA routers can be divided into two groups. Combined global-detailed routers [85, 90, 91, 92, 93, 94, 95, 96] determine a complete routing path in one step, while two-step routing algorithms first perform global routing [97, 98] to determine which logic block pins and channel segments each net will use, and then perform detailed routing [99, 100, 101, 34] to determine the wire(s) each net will use within each of the specified channel segments. A channel segment is the length of routing channel that spans one logic block — a channel that spans M logic blocks contains M channel segments. The task of an FPGA detailed router is often difficult or impossible because FPGA routing has limited flexibility and the detailed router is highly constrained by the decisions the global router made about which channel segments each net must use. Combined global-detailed routers have the potential to more fully optimize the routing, since they are free of such constraints.

Of the routers listed above, only those of [85, 95, 94] use timing analysis (see Section 2.2.5) to determine which nets are on, or "almost" on, the critical path so they can be given priority for fast routing paths. Since much of our work is concerned with timing-driven routing, we will focus on these three algorithms. At their core, all these routers use variants of maze routers [102] to connect the terminals of each net. A maze router essentially consists of running Dijkstra's algorithm [103] to find the shortest (lowest total cost) path between a net source node and a net sink node in a routing-resource graph. All of these algorithms perform multiple routing iterations in which some or all of the nets are ripped-up and rerouted by different paths to resolve competition for routing resources or improve circuit speed. Both [94] and [95] use timing analysis only to help identify good nets to rip-up and re-route — nets which are likely to lead to a circuit speed-up if they can be rerouted using a faster path. Ripping-up and re-routing these nets only changes the net ordering, however; they are all routed by the same maze routing algorithm, regardless of how timing-critical they are. The PathFinder negotiated congestion-delay algorithm [85] uses a more sophisticated technique in which the congestion-delay trade-off of each connection is controlled by how timing critical it is. In other words, a timing-critical connection will be routed by a minimum delay path even if it is congested, while a non-timing-critical net will take a longer, uncongested path. This algorithm produces excellent results and incorporates several important ideas, so we describe it in detail below.

Pathfinder repeatedly rips-up and re-routes every net in the circuit until all congestion is resolved — this idea is due to Nair [104]. Ripping-up and re-routing every net in the circuit once is called a *routing iteration.* During the first routing iteration, every connection is routed for minimum delay, *even if this leads to congestion*, or overuse, of some routing resources. A circuit routing in which some routing resource is overused, such as a wire being used by two different nets, is not a legal routing. Consequently, when overuse exists at the end of a routing iteration, another routing iteration (or more) must be performed to resolve this congestion. After each routing iteration the cost of overusing a routing resource is increased, so that the probability of resolving all congestion increases. At the end of each routing iteration we have a complete, but potentially somewhat illegal, routing. We can therefore determine the net delays from this routing, and perform a full timing analysis to compute the slack (see Section 2.2.5) of each source-sink connection. These slack values are used in the next routing iteration to control how much attention each connection pays to delay, and how much is paid to congestion-avoidance. Pseudo-code for the algorithm is given in Figure 2.10.

The criticality of the connection from the source of net i to one of its sinks, j, is:

$$Crit(i, j) = 1 - \frac{slack(i, j)}{D_{max}} \qquad (2.4)$$

where D_{max} is the delay of the circuit critical path, and slack(i,j) is the amount of delay that could be added to this connection before it affected the circuit's critical path. Crit(i,j) is therefore between 0 and 1.

The cost of using a routing resource node, n, as part of connection (i,j) is

$$Cost(n) = Crit(i, j) \cdot delay(n) + [1 - Crit(i, j)] \cdot [b(n) + h(n)] \cdot p(n). \qquad (2.5)$$

The first term in (2.5) is the delay sensitive term — the criticality of the connection times the intrinsic delay of the node. The second term is the congestion sensitive term. b(n) is the base cost of node n, and is set equal to delay(n) in [85]. h(n) is the historical congestion of node n; it is increased after every routing iteration in which node n is overused and gives the router "congestion memory." p(n) is the present congestion cost of node n; it is 1 if using this node to route the current connection will not cause any overuse, and increases with the amount of overuse of the node. p(n) is also a function of the number of routing iterations that have been performed. In early iterations, p(n) grows slowly with the current overuse of node n; in later iterations, p(n) goes up very rapidly with overuse of node n.

```
Let: RT(i) be the set of nodes, n, in the current routing of net(i).

Crit(i,j) = 1 for all nets i and sinks j;
while (overused resources exist) {    /* Illegal routing? */

    for (each net, i) {
        rip-up routing tree RT(i) and update affected p(n) values;
        RT(i) = NetSource(i);

        for (each sink, j, of net(i) in decreasing crit(i,j) order) {
            PriorityQueue = RT(i) at PathCost(n) = crit(i,j)·delay(n) for
                            each node n in RT(i);
            while (sink(i,j) not found) {
                Remove lowest cost node, m, from PriorityQueue;
                for (all fanout nodes n of node m) {
                    Add n to PriorityQueue at PathCost(n) =
                        Cost(n) + PathCost(m);
                }
            }

            for (all nodes, n, in path from RT(i) to sink(i,j)) { /* Backtrace */
                Update p(n);
                Add n to RT(i);
            }
        }
    }

    Update h(n) for all n;
    Perform timing analysis and update Crit(i,j) for all nets i and sinks j;
} /* End of one routing iteration */
```

FIGURE 2.10 Pseudo-code of the Pathfinder routing algorithm.

The excellent performance of Pathfinder is due to two innovations: allowing overuse of routing resources, and using the cost function of (2.5) to allow congestion to gradually be resolved, and timing to be directly optimized. By slowly increasing the cost of congestion, via p(n) and h(n), as more routing iterations are performed, connections that are on or near the critical path tend to take the fastest paths and stay there, while less timing-critical connections are gradually forced off any overused nodes onto slower paths.

Notice that the router of Figure 2.10 uses a breadth-first search through the routing resource graph to connect net terminals. The creators of Pathfinder [85] also describe

an enhancement to the basic algorithm that uses an A*, or directed, search [105] to speed execution. In Figure 2.10 a node is added to the priority queue with a PathCost equal to the sum of all the node costs along the path up to and including it. To convert this to an A* search, one simply adds this term to a lower bound on the sum of the node costs needed to reach the target sink from the current node; the result is then used as the sort value when the node is added to the priority queue.

Finally, Ebeling et al [85] also describe a purely routability-driven variant of the Pathfinder router, which they call the Pathfinder negotiated congestion algorithm. This algorithm simply sets the criticality of every net, Crit(i,j), to 0 so that the cost of a node is given solely by the congestion-sensitive term in (2.5). As well, this router connects the current routing tree to the first net sink found, rather than a pre-determined target sink, during maze expansion.

The router used in the simultaneous placement and routing tool developed by Nag and Rutenbar [86] has one unique feature of interest. This router is again maze-router based, and it is targeted at Xilinx XC4000 series FPGAs. When routing a multi-terminal net, a maze router will typically route to the closest sink, and then use this partial routing as the source (start point) when it attempts to route to the next closest sink, and so on. This can cause problems when routing high-fanout nets on FPGAs that contain some long wire segments. High-fanout nets typically span most of the FPGA, but the distance from one sink to the next closest sink is usually only a few logic blocks. Consequently, a traditional maze router, that looks only at how to connect the partial routing to the next closest sink, will tend to use short wires to build the routing trees for high-fanout nets, even though using longer wires would be more efficient and result in faster nets. Nag solved this problem by dramatically reducing the cost of using a long line when the net bounding box spanned most of the FPGA. The router then saw the cost of using a long wire to be less than that of even one short wire, so it would use long lines to connect even to nearby sinks and construct a routing tree of long wires for high-fanout nets. This idea of varying the cost of resources depending on the type of connection being routed, or *dynamic weighting*, is a powerful one; in some sense Pathfinder's varying the weighting of delay and congestion according to the criticality of the connection being routed is another example of this idea.

One shortcoming of current non-commercial timing-driven FPGA routers is that they are designed to optimize only the linear delay model, in which every routing resource has a fixed delay.[1] Most FPGAs contain at least some pass transistor switches in their routing, so the delay of a routing resource actually depends on the topology of the net

1. SEGA can use a more advanced delay model, but [36] showed that SEGA achieved better speed with a cost function that emphasized recombining the two-point nets passed to it by the global router into multi-terminal nets than it did with this delay-based cost function.

using it. As well, since the FPGAs we study in Chapter 7 include buffers in the routing, the router must understand when to use a buffered switch and when to use a pass transistor. Unfortunately, no non-commercial FPGA router is "buffer-aware." The Xilinx commercial router, described by Trimberger in [106], is buffer-aware and uses the Penfield-Rubinstein [107] delay model during routing. It is likely that the routers of other FPGA companies whose products contain a mix of pass transistors and buffers are also buffer-aware, but to our knowledge the algorithms used by these other companies have never been made public.

Considerable work has been done in the standard cell and MPGA routing areas on routing under more accurate delay models [108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121] and buffer insertion [122, 123, 124, 125, 126]. However, much of this work is not easily applicable to the FPGA routing problem because:

1. MPGA and standard cell routers can choose exactly where they want wires, how long these wires should be, and where buffers should be placed. Since all FPGA routing resources are prefabricated, however, FPGA routers are constrained to choose from a set of prefabricated wires and switches. The flexibility of FPGA routing is low enough that if the router decides to connect two wires together it usually has no further choice about whether to insert a buffer or not at the join. There is usually only one switch that can connect these two wires, and whether it is a buffer or a pass transistor was determined when the FPGA was fabricated. In graph theoretic terms, FPGA routing consists of finding Steiner trees embedded in a graph, while MPGA routing consists of finding Steiner trees embedded in the Manhattan plane [90].
2. The complexity of many MPGA and standard cell routing algorithms is quite high — $O(k^3)$ to $O(k^4)$ is common, where k is the number of terminals on a net. Since we will be routing circuits with thousands of nets, and a few of these nets have hundreds of terminals, we must use algorithms with reasonably low (ideally linear) complexity [127].

Nonetheless, some ideas from the MPGA world are relevant to our work. For example, in the absence of congestion, PathFinder attempts to greedily optimize a combination of wirelength and (linear delay model) delay to a net's sinks, which is similar to the approach of Alpert et al in [110]. In a similar vein, [109] and [111] attempt to greedily optimize a combination of the Elmore delay and wirelength; it should be possible to adapt these algorithms to FPGAs. The time complexity of these algorithms is quite high, however. The algorithm of [111] is $O(k^4)$, and while the authors do not give the complexity of their algorithm in [109], it appears to be at least $O(k^3)$. Since adapting these algorithms to FPGAs involves routing within a large routing-resource graph, their complexity may increase even further and make them impractical.

2.2.4 Delay Modelling

One must compute the delay of a route from a net source to any of its sinks in order to:

1. Determine the speed of a circuit after it has been routed, and
2. Determine the delay of different net topologies during routing.

Ideally, one would use a circuit simulator such as SPICE to obtain highly accurate delay estimates, but the CPU time required to run SPICE on the thousands of nets in a typical circuit is prohibitive. Instead, previous researchers have modelled pass transistors as linear resistors and wires as an RC pi-network, so that a net's routing may be modelled as an RC-tree [22]. In [22], the Penfield-Rubinstein delay model [107] was then used to determine an upper and lower bound on the delay of the RC-tree to each of the net sinks. An alternative to the Penfield-Rubinstein model is the Elmore delay [128], which is the most widely used delay estimate in routing research [108]. The Elmore delay was originally defined only for RC-trees, but it has been combined with a common model of buffer delay to allow its use with circuits that contain buffers, as well as resistors and capacitors [112]. Each buffer is modelled by a constant delay and a resistor. The constant delay accounts for the intrinsic delay of the buffer, while the resistance accounts for the load-dependence of the buffer delay. Figure 2.11 shows the RC-model for each of the three elements of FPGA routing. Note that pass transistors and buffers attached to a wire add parasitic capacitance regardless of whether they are on or off.[1]

The Elmore delay of a source-sink path is then [112]:

$$\sum_{i \in \text{ Source-sink path}} R_i \cdot C(subtree_i) + T_{d,i} \qquad (2.6)$$

where $T_{d,i}$ is the intrinsic delay of a buffer if element i is a buffer, and 0 otherwise. R_i is the equivalent resistance of element i (R_{wire}, R_{buf}, or R_{pass}). In (2.6), $C(subtree_i)$ is the total capacitance of the dc-connected subtree rooted at element i — that is the total downstream capacitance which is not isolated by buffers.

1. Notice that we model the capacitance of both "on" and "off" pass transistors as being purely due to diffusion capacitance. In fact the capacitance of an "on" pass transistor is larger than that of an "off" pass transistor, since the channel created when a transistor is "on" has capacitance to the gate and to the substrate. Since relatively few pass transistors are "on" at any time and most of the capacitance in an FPGA is metal capacitance, the error in total capacitance caused by this approximation is small (~1% to 2%).

4.3 Routability-Driven Router

Recall that VPR incorporates two different routers: one that is purely routability-driven, and one that is both routability and timing-driven. Both these routers can perform either combined global-detailed routing or global routing alone simply by changing the routing-resource graph passed to them. In this section we describe a purely routability-driven router, while the next section describes the timing-driven router. Once a routing is complete, VPR's graphics can be used to examine it — see Appendix A for sample pictures.

4.3.1 Cost Functions and Routing Schedules

Our routability-driven router is based on the Pathfinder negotiated congestion algorithm [85]; this purely routability-driven variant of the Pathfinder algorithm was described in Section 2.2.3. In the discussion below we will focus on new enhancements in our router and on important portions of the algorithm implementation that were not described in [85].

We define the cost of a node somewhat differently than [85] (see Equation (2.5)); the cost of using routing resource n when it is reached by connecting it to routing resource m is:

$$Cost(n) = b(n) \cdot h(n) \cdot p(n) + BendCost(n, m), \quad (4.3)$$

where the b(n), h(n) and p(n) are the base cost, historical congestion, and present congestion terms defined in Section 2.2.3. The BendCost(n,m) term is an enhancement we have made to improve the results of global routing. It penalizes bends when global routing is being performed, since global routes with many bends make it difficult or impossible for a subsequent detailed routing phase to utilize long wire segments [36, 31]. Hence reducing the number of bends in a global routing tends to lead to detailed routes that are both faster and require fewer tracks. If global routing only is being performed, BendCost(n,m) is 1 if making the connection from node m to node n implies a bend — i.e. node m is a horizontal channel segment and node n is a vertical channel segment or vice versa. Including this BendCost term in the total cost of using a node produces routes with very few unnecessary bends and does not significantly increase the global routing track count. If combined global-detailed routing is being performed there is no need to penalize bends, so BendCost(n,m) is always zero in this case.

Notice that the functional form of (4.3) is different than that of (2.5); we multiply b(n) and h(n) together rather than adding them. When adding terms in cost functions, it is important to ensure they are properly normalized to roughly the same range of magnitude so that both terms have an effect. We avoid having to normalize h(n) to b(n) by converting the addition to a multiplication.

In [85], the base cost of a node, b(n) was set to its intrinsic delay. We have found that this is not the best choice; on average, about 10% fewer tracks per channel are required when the base costs of Table 4.1 are used instead. Note that the performance of the router is not extremely sensitive to the exact base costs chosen; the congestion avoidance terms in (4.3) ensure that the primary goal of the router is congestion avoidance, regardless of the b(n) values.

TABLE 4.1 Base costs of different types of routing resource.

Routing Resource, n	Base Cost, b(n)
Wire segment	1
Logic block output pin	1
Logic block input pin	0.95
Source	1
Sink	0

Four of the five b(n) values in Table 4.1 have virtually the same value — this encourages the router to use as few of these resources as possible to route each connection. The b(n) value for an logic block input pin and for a sink are set to less than 1 to save CPU time. Since the maze expansion used to route a connection terminates when it reaches a sink corresponding to one of the net terminals, some CPU savings can be obtained by costing resources so that sinks tend to be reached earlier in the maze expansion. In other words, we would like to cost logic block input pins and sinks so that the maze expansion checks if the logic block next to a wire segment contains one of the net sinks for which we are searching before it expands more wire segments. To achieve this behaviour, a logic block input pin has a base cost of slightly less than 1, and a sink has a base cost of zero. Since congestion can not occur at sinks, using a base cost of zero for them will not cause route failures. Using a lower cost for logic block inputs and sinks in this way speeds the routability-driven router up by 1.5 to 2 times, depending on the FPGA architecture and circuit being routed.

We experimented with several different possibilities for the base costs of wire segments of different lengths: intrinsic delay, 1, length, the square root of length, and length+1. For both the routability-driven router described in this section, and the timing-driven router of the next section, setting b(n) to 1 regardless of a wire segment's length yields the best results. Using the intrinsic delay of each routing resource as its

b(n) value increased the number of tracks required per channel by 10% on average. The other choices of b(n) led to performance between these extremes. Setting the cost of a wire segment to 1 regardless of its length also led to the highest speed circuits for the routability-driven router, since it encourages connections to use the smallest number of routing resources possible.

Ebeling et al [85] did not describe the exact functional form of the h(n) and p(n) congestion avoidance terms in (4.3), so we list the equations we use below. The present congestion penalty is updated whenever any net is ripped-up and re-routed according to

$$p(n) = 1 + max(0, [occupancy(n) + 1 - capacity(n)] \cdot p_{fac}), \quad (4.4)$$

where occupancy(n) is the number of nets currently using routing resource n, and capacity(n) is the maximum number of nets that can legally use node n. The historical congestion penalty is updated only after an entire routing iteration. Its value during routing iteration i is:

$$h(n)^i = \begin{cases} 1, & i = 1 \\ h(n)^{i-1} + max(0, [occupancy(n) - capacity(n)] \cdot h_{fac}), & i > 1 \end{cases} \quad (4.5)$$

The values of h_{fac} and p_{fac} in each routing iteration define what we call the *routing schedule* [2]. We have found that it is sufficient to keep h_{fac} constant for all routing iterations; the fact that h(n) is incremented after every iteration in which node n is overused provides sufficient increase in the historical congestion penalty. Any value of h_{fac} between 0.2 and 1 works equally well. To achieve the absolute highest quality results, p_{fac} should initially be small, allowing congestion with little penalty, and gradually increase from iteration to iteration. For these highest quality results, p_{fac} should be 0.5 or less in the first iteration, and 1.5 to 2 times its previous value in each subsequent iteration. When the cost of congestion is increased this gradually, however, it takes several routing iterations (typically 5 - 10) to route even fairly "easy" routing problems, where there is more routing available than is needed by the circuit. For such "easy" problems, the router can be sped up by a factor of two to three times by making congestion very expensive immediately — in this case, we make p_{fac} 10000 in the first routing iteration, ensuring the router will always avoid congestion if it can, even in the first routing iteration. The amount of quality sacrificed by making congestion expensive immediately is quite small — typically the minimum achievable track count increases by only 2% to 4%. Throughout this book we will use the slowly increasing congestion cost schedule to achieve the absolute highest quality, however.

If a circuit has not routed in 30 routing iterations we normally classify it as unroutable. Allowing the router to try 45 routing iterations before giving up reduces the minimum track count by only 1% to 4%, on average.

4.3.2 Speed Enhancements

In addition to tuning the routing schedule and optimizing the routing-resource base costs, we made two other speed enhancements to the Pathfinder negotiated-congestion algorithm. First, we do not allow net routings to go more than three channels outside of their net bounding box. Since our routability-driven router uses a breadth-first maze router to make connections, this enhancement significantly reduces CPU time. Restricting each route to remain within 3 channels of its bounding box had no noticeable effect on the quality of the routing.

The second speed enhancement aids the routing of high-fanout nets, and results in an order-of-magnitude router speedup on large circuits. Recall that to route a k-terminal net, the maze router contained in VPR is invoked k-1 times. In the first invocation, the maze routing wavefront expands out from the net source until it reaches any one of the k-1 net sinks. The path from source to sink is now the first part of this net's routing. In a traditional maze router, and in the Pathfinder algorithm, the maze expansion (the PriorityQueue in Figure 2.10) is emptied, and a new wavefront expansion is started from the entire net routing found thus far. After k-1 invocations of the maze router all k terminals of the net will be connected.

This approach requires considerable CPU time for high-fanout nets. High-fanout nets usually span most or all of the FPGA. Therefore, in the latter invocations of the maze router the partial routing used as the net source will be very large, and it will take a long time to expand the maze router wavefront out to the next sink. We have developed a more efficient method. When a net sink is reached, we add all the routing resource segments required to connect the sink and the current partial routing to the wavefront (i.e. the PriorityQueue) with a cost of 0. We do not empty the current maze routing wavefront, but instead continue expanding normally. Since the new path added to the partial routing has a cost of zero, the maze router will expand around it first. Since this new path is typically fairly small, it will take relatively little time to add this new wavefront, and the next sink will be reached much more quickly than if the entire wavefront expansion had been restarted. Figure 4.11 illustrates the difference graphically. The shortest path computed to most of the nodes in the wavefront is still valid after a net sink is reached; we are taking advantage of this by restarting only a local portion of the wavefront instead of restarting the entire wavefront.