PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs

Larry McMurchie and Carl Ebeling
Dept. of Computer Science and Engineering
University of Washington, Seattle, WA

Abstract

Routing FPGAs is a challenging problem because of the relative scarcity of routing resources, both wires and connection points. This can lead either to slow implementations caused by long wiring paths that avoid congestion or a failure to route all signals. This paper presents PathFinder, a router that balances the goals of performance and routability. PathFinder uses an iterative algorithm that converges to a solution in which all signals are routed while achieving close to the optimal performance allowed by the placement. Routability is achieved by forcing signals to negotiate for a resource and thereby determine which signal needs the resource most. Delay is minimized by allowing the more critical signals a greater say in this negotiation. Because PathFinder requires only a directed graph to describe the architecture of routing resources, it adapts readily to a wide variety of FPGA architectures such as Triptych, Xilinx 3000 and mesh-connected arrays of FPGAs. The results of routing ISCAS benchmarks on the Triptych FPGA architecture show an average increase of only 4.5% in critical path delay over the optimum delay for a placement. Routes of ISCAS benchmarks on the Xilinx 3000 architecture show a greater completion rate than commercial tools, as well as 11% faster implementations.

1 Introduction

The problem of routing FPGAs can be stated simply as that of assigning signals to routing resources in order to successfully route all signals while achieving a given overall performance. The first goal, complete routing of all signals, is difficult to achieve in FPGAs because of the lack of routing resources. The usual approach to achieving this goal is to minimize the use of routing resources by constructing minimum routing trees for each signal. Although this reduces the demand for routing resources, signals will still compete for the same resources and 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 in general are much more expensive in terms of routing resources than minimum routing trees. Thus the solution to the entire routing problem requires the simultaneous solution to two interacting and competing subproblems.

The problem of routing FPGAs bears a considerable resemblance to the problem of global routing for custom integrated circuit design and one would hope to be able to apply the same algorithms to FPGAs. However, the two problems differ in several fundamental respects. First, routing resources in FPGAs are discrete and scarce, while they are reasonably continuous in custom integrated circuits. For this reason FPGAs require an integrated approach using both global and detailed routing. A second difference is that the global routing problem for custom ICs is rooted in an undirected graph embedded in Cartesian space. In FPGAs the switches are often directional, and the routing resources connect arbitrary (but fixed) locations, requiring a directed graph which may not be embedded in Cartesian space. Both of these distinctions are important, as they prevent direct application of much of the work that has been done in custom IC routing to FPGAs.

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, which must be rerouted by hand. A plethora of rip-up and retry approaches have been proposed to remedy the deficiencies of this approach ([Dees81], [Linsker84], [Cohn91]). The basic problem with rip-up and retry is that the success of a route is dependent not just on the choice of which nets to reroute, but also on the order in which rerouting is done.

Several papers have described versions of shortest path with rip-up and retry targeted to FPGAs. [Hill91] uses a breadth-first search while performing routes in random order and a "blame factor" is introduced to decide what routes need to be ripped up when a connection is unrealized. [Brown92] uses a global router to assign connections so that channel densities are balanced; a detailed router generates families of explicit paths within channels to resolve congestion. If some connections are unrealizable, the channel routes are ripped up and a rerouting is performed using larger families of paths.

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 ([Brown92]). How to optimally balance the competing goals of minimizing delay of critical paths and eliminating congestion is an open question. [Alexander94] presents a general multi-weighted graph formalism that attempts to accommodate delay and congestion. Results based only upon congestion elimination compare favorably with those of [Brown92]; however, the work is preliminary and no results are given that compare critical path lengths.

The most extensive work to date factoring delay into FPGA routing has been that of [Frankle92]. In this work a slack analysis is performed to calculate upper bounds for individual source/sink connections. A rip-up and retry scheme routes signals, increasing upper bounds as needed. Once the routing has completed, selected connections are rerouted so as to reduce the overall delay. Although the results of this scheme are good (delays of the final routes average 16% higher than optimal), the performance of this scheme, as in the other rip-up and retry approaches, suffers from a dependency upon the order in which connections are routed. Also, by performing a slack analysis only at the beginning and the end of the routing process, opportunities for balancing congestion and delay are lost.

In this paper we present PathFinder, an iterative algorithm that balances the competing goals of eliminating congestion and minimizing delay of critical paths in an iterative framework. In this framework, signals are allowed to share routing resources initially, but subsequently must negotiate with other signals to determine which signal needs the shared resource most. A timing analysis is performed every iteration to apply pressure continuously to routes that can potentially become critical if left unchecked. The emphasis of our approach is to adjust the costs of routing resources in a gradual, semi-equilibrium fashion to achieve an optimum distribution of resources.

We have been careful not to introduce architecture-specific features into PathFinder so that it can be applied to any FPGA architecture. The router is customized to a specific architecture by specifying a template that describes routing resources and pin permutability. Although PathFinder could be improved by adding architecture-specific knowledge, our experiments show that it performs extremely well even without this extra knowledge.

2 General Approach

PathFinder is derived from an iterative scheme for the global routing of custom IC's developed by Nair [Nair87]. This scheme differs in several aspects from most forms of rip-up and retry. Only one net is ripped up at a time, but every net is ripped up and rerouted on every iteration, even if the net does not pass through a congested area. In this way nets passing through uncongested areas can be diverted to make room for other nets currently in congested regions. Nets are ripped up and rerouted in the same order every iteration. Our routing algorithm differs from Nair's primarily in the construction of the cost function and the handling of delay.

PathFinder is composed of two parts: a signal router, which routes one signal at a time using a shortest-path algorithm, and a global router, which calls the signal router to route all signals, adjusting the resource costs in order to achieve a complete routing. The signal router uses a breadth-first search to find the shortest path given a congestion cost and delay for each routing resource. The global router dynamically adjusts the congestion penalty of each routing resource based on the demands signals place on that resource. During the first iteration of the global router there is no cost for sharing routing resources and individual routing resources may be used by more than one signal. However, during subsequent iterations the penalty is gradually increased so that signals in effect negotiate for resources. Signals may use shared resources that are in high demand if all alternative routes utilize resources in even higher demand; other signals will tend to spread out and use resources in lower demand. The global router reroutes signals using the signal router until no more resources are shared. The use of a cost function that *gradually* increases the penalty for sharing is a significant departure from Nair's algorithm, which assigns a cost of infinity to resources whose capacity is exceeded.

In addition to minimizing congestion, the signal router ensures that the delays of all signal paths stay within the critical path delay. For multiple sinks, low congestion cost can be achieved by a minimum Steiner tree, but this can result in long delays. Low delay can be achieved by a minimum-delay tree, but this may mean competition by many signals for the same routing resources. To achieve a balance, the signal router uses the relative contribution of each connection in the circuit (i.e. source-sink pair) to the overall delay of the circuit to determine how to trade off congestion and delay. A slack ratio is computed for each connection in the circuit as the ratio of the delay of the longest path using that connection to the delay of the circuit's longest (i.e. most critical) path. Thus, every connection on the longest path has a slack ratio of 1, while connections on the least critical paths have slack ratios close to 0. The inverse of the slack ratio gives the factor by which the delay of a path can be expanded before the circuit is slowed down.

The key idea behind the signal router is that connections with a slack ratio close to 1 will be assigned greater weight in negotiating for resources and consequently will be routed directly (i.e. using a minimum-delay route) from source to sink. Connections with a small slack ratio will have less weight and pay more attention to congestion-avoidance during routing. A net with multiple sinks (which corresponds to several connections with varying slack ratios) will be routed using a combined strategy, and will not be constrained to either an overall minimum Steiner tree or minimum-delay tree route. The slack mechanism provides a smooth tradeoff between these two extremes.

3 PathFinder Algorithm

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 to the switches that connect these nodes. Associated with each node n in the architecture is a constant delay d_n and a congestion cost c_n determined by the competition among signals for 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 s_i with all its t_{ij}.

3.1 Negotiated Congestion Router

We will present the Negotiated Congestion (NC) algorithm in this section, and then extend it to minimize delay in the next section. The cost of using a given node n in a route is given by

$$c_n = (b_n + h_n) * p_n \qquad (1)$$

where b_n is the base cost of using n, h_n is related to the history of congestion on n during previous iterations of the global router, and p_n is related to the number of other signals presently using n. A reasonable choice for b_n is the intrinsic delay d_n of the node n since minimizing the delay of a path in general minimizes the routing resources of a path. In the remainder of this paper we set $b_n = d_n$.

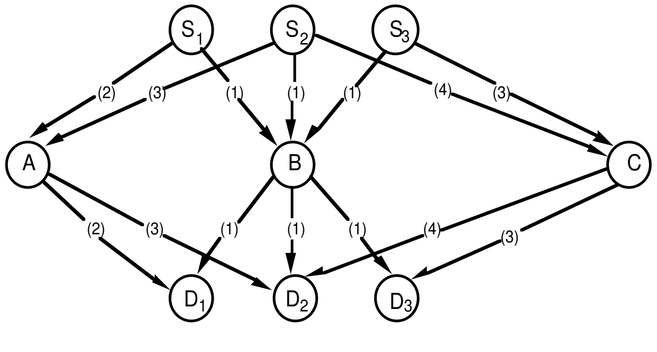

Figure 1. First order congestion

The h_n and p_n terms are motivated by the routing problems in Figures 1 and 2. Figure 1 shows a *first order congestion* problem. We need to route signals 1, 2, and 3 from their sources S_1, S_2, and S3 to their respective sinks D_1, D_2, and D_3. The arcs in the graph represent partial paths, with the associated costs in parentheses. Ignoring congestion, the minimum cost path for each signal would use node B. If a simple obstacle avoidance routing scheme is used, the order in which the signals are routed becomes important. Some orderings will not be successfully routed. Moreover, the total routing cost will be a minimum only if we start with signal 2.

In the NC algorithm, the first-order congestion of Figure 1 is solved using the p_n factor in our cost function (assuming for the time being $h_n = 0$). During the first iteration of the global router, p_n is initialized to one, thus no penalty is imposed for the use of n regardless of how many signals occupy n. During subsequent iterations, this penalty is gradually increased, depending on how many signals share n. In the first iteration therefore, all three signals share B. During some later iteration signal 1 will find that a route through A gives a lower cost than through the congested node B. During an even later iteration signal 3 will find that a route through C gives a lower cost than 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 would become important.

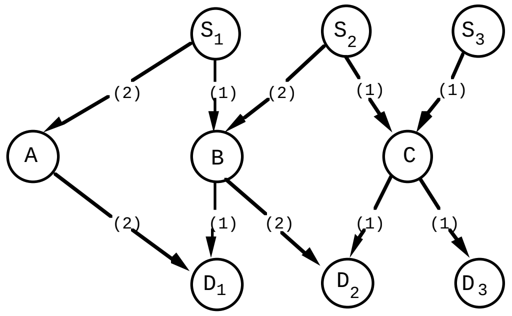

Figure 2. Second order congestion

Figure 2 shows an example of *second order congestion*. 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 B, and signals 2 and 3 share node C. For ripup 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 will be difficult in general.

This second-order congestion problem cannot be solved using p_n alone. The term h_n is required to successfully route this problem. Each iteration that node C is shared, h_n is increased slightly. After enough iterations, the route through C will become more expensive for signal 2 than the route through B. Once B is shared by both signals 1 and 2, signal 1 will be rerouted through A, and the congestion will be eliminated. The effect of h_n is to permanently increase the cost of using congested nodes so that routes through other nodes are attempted. The addition of this term to account for the history of congestion of a node is another distinction between the NC algorithm and Nair's.

The details of the NC algorithm are given below. The signal router loop starts at step 2. The routing tree RT_i from the previous global routing iteration is erased and initialized to the signal source. A loop over all sinks t_{ij} of this signal is begun at step 5. A breadth-first search for the closest sink t_{ij} is performed using the priority queue PQ in steps 7-12. Fanouts n of node m are added to the priority queue at $c_n + P_{im}$, where P_{im} is the cost of the path from s_i to node m.

After a sink is found, all nodes along a backtraced path from the sink to source are added to RT_i (steps 13-16), and this updated RT_i is the source for the search for the next sink (step 6). In this way, all locations on routes to previously-found sinks are used as potential sources for routes to subsequent sinks. This is similar to Prim's algorithm for determining a minimum spanning tree over an undirected graph. This algorithm for constructing the routing tree is identical to an algorithm suggested by [Takahishi80] 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 global router to be successful in adjusting costs to eliminate congestion.

Algorithm: Negotiated Congestion (NC)

```
While shared resources exist (global router)                  [1]
   Loop over all signals $i$ (signal router)                   [2]
      Rip up routing tree $RT_i$                               [3]
      $RT_i$ <- $s_i$                                          [4]
      Loop until all sinks $t_{ij}$ have been found            [5]
         Initialize priority queue $PQ$ to $RT_i$ at cost 0    [6]
         Loop until new $t_{ij}$ is found                      [7]
            Remove lowest cost node $m$ from $PQ$              [8]
            Loop over fanouts $n$ of node $m$                  [9]
               Add $n$ to $PQ$ at cost $c_n + P_{im}$          [10]
         End                                                   [11]
      End                                                      [12]
      Loop over nodes $n$ in path $t_{ij}$ to $s_i$ (backtrace) [13]
         Update $c_n$                                          [14]
         Add $n$ to $RT_i$                                     [15]
      End                                                      [16]
     End                                                       [17]
   End                                                         [18]
End                                                            [19]
```

3.2 Negotiated Congestion/Delay Router

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

$$C_n = A_{ij}\, d_n + (1 - A_{ij})\, c_n \qquad (2)$$

where c_n is defined in eq. (1) and A_{ij} is the slack ratio

$$A_{ij} = D_{ij} / D_{max} \qquad (3)$$

where D_{ij} is the longest path containing the arc (s_i, t_{ij}), and D_{max} is the maximum over all paths, that is, the critical path delay. Thus, $0 < A_{ij} \leq 1$.

The first term of eq. (2) is the delay-sensitive term, while the second term is congestion-based. Equations. (2) and (3) 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 it sees for using node n is simply the delay term; hence, a minimum-delay route will be used and congestion will be ignored. If a source-sink pair belongs to a path whose delay is much smaller than the critical path, its A_{ij} will be small and the congestion term will dominate, resulting in a route which avoids congestion at the expense of extra delay.

To accommodate delay, the NC algorithm is changed as follows. First, the A_{ij} are initialized to 1. Thus during the first iteration the global router finds the minimum-delay route for every signal. The A_{ij} are recomputed each subsequent iteration. Second, the sinks are routed in decreasing A_{ij} order. Third, the priority queue (line [6]) is initialized to RT_i at cost $A_{ij}\, dj$. The net effect of this initialization is that nodes that are already in the (partial) routing tree will have only a delay component. We shall refer to these modifications as the NCD algorithm, or simply PathFinder, in the remainder of this paper.

The global router completes when no more shared resources exist. Note that by recalculating the A_{ij} , we have kept a tight reign on the critical path. Over the course of iterations, the critical path increases only to the extent required to resolve congestion. This approach is fundamentally different from other schemes ([Brown92], [Frankle92]) which attempt to resolve congestion first, then reduce delay by rerouting critical nets.

3.4 Delay Bounds

In this section we show that if h_n is bounded by d_n, then algorithm NCD guarantees a worst case path delay equal to the minimum delay route of the critical path. That is, in this situation algorithm NCD achieves the fastest implementation allowed by the placement. In practice, h_n is allowed to increase gradually until a complete route is found. For very congested circuits, h_n will exceed d_n, but as we show experimentally in Section 4, algorithm NCD comes very close to this bound in practice.

Theorem 1 If $h_n <= d_n$ for all nodes, then the delay of any signal path routed by algorithm NCD is bounded by D_{max}, the delay of the longest minimum-delay path in the circuit.

Proof sketch: When algorithm NCD terminates successfully, the p_n term in equation 2 is 1 and can be ignored. Let R be the most critical routed path and S be the shortest delay route for R. The cost of S is given by:

$$\begin{aligned} C_S &= \sum_{n \in S} c_n \\ &= \sum_{n \in S} (A_{ij} d_n + (1 - A_{ij})(d_n + h_n)) \\ &= \sum_{n \in S} d_n + \sum_{n \in S} (1 - A_{ij}) h_n \end{aligned}$$

Since $h_n \leq d_n$,

$$\begin{aligned} C_S &\leq D_{ij} + \sum_{n \in S} (1 - A_{ij}) d_n \\ &\leq D_{ij} + (1 - A_{ij}) D_{ij} \\ &\leq D_{ij} + (D_{MAX} - D_{ij}) D_{ij} / D_{MAX} \end{aligned}$$

Since $D_{ij} / D_{MAX} \leq 1$,

$$\begin{aligned} C_S &\leq D_{ij} + D_{MAX} - D_{ij} \\ &\leq D_{MAX} \end{aligned}$$

The cost of R must be less than the cost of S, thus the delay of R must be less than the cost of S, which is less than D_{max}.

3.5 Enhancements

Several enhancements can increase the speed of the NCD algorithm without adversely affecting the quality of the route. One enhancement is to introduce the A* algorithm into the breadth-first search loop. A* uses lower bounds on path lengths to bound the breadth-first search. A* can be applied to the congestion/delay router by tabulating the cost of minimum-delay routes from every node to all the potential sinks. During the first iteration, the search can be made linear in the number of nodes along a minimum-delay path. As iterations progress, increasing p_n and h_n cause this lower bound to prove less and less useful, and the search expands. As a result, the CPU time grows, but still remains less than a full breadth-first search.

Another enhancement is to route only the signals involved in congested nodes. This is a significant departure from Nair's original algorithm. If one examines the routing problems in Figures 1 and 2 with this modification, the same result is obtained as if one had rerouted all signals during every iteration. To date we have not seen any cases where routing only congested nodes resulted in a lower-quality route. In our experience the number of iterations increases, but the total running time decreases.

4 Results

We have performed experiments using PathFinder (specifically algorithm NCD) on two different FPGA architectures. We chose Triptych because the limited routing resources would expose the limitations of the algorithm, and Xilinx because this allowed comparison to an FPGA router currently in wide use.

For both architectures the routing resources were described using the schematic capture system WireC. The output of the WireC system is a directed graph over all the routing resources. All architectural information required by PathFinder including delay information is contained in this directed graph. Retargeting PathFinder to a new architecture is a straightforward matter of modifying an existing template or creating a new one; no code modifications to the router are required. This approach provides a convenient mechanism for changing configurations of routing resources and examining the impact of these changes on the routability of circuits.

4.1 Experiments on Triptych

The Triptych architecture is an array of 3-input blocks ([Hauck92]). These blocks, known as RLBs (Routing and Logic Blocks) contain 3-input LUTs, as well as routing resources that can route inputs through blocks to neighboring blocks or onto buses. This approach is markedly different from other FPGA architectures, notably Xilinx, which place CLBs in a sea of routing resources. By comparison, Triptych has considerably fewer routing resources, many of which connect only nearest neighbors. The placement problem is obviously coupled closely to the routing problem. A placement program was constructed using a simulated annealing approach, where the cost function is composed of both a routing distance metric and a metric that attempts to estimate routing congestion. Even with these measures of routability included in the placement cost function, PathFinder has the difficult problem of allocating the relatively limited routing resources to signals to achieve feasible source-sink routes. Factoring in the delay of critical paths obviously complicates the problem.

Bench	Reps	Logic Levels	Optimal Delay	Routed Delay	% over optimal
1	10	2	23.3	23.3	0.0%
2	4	6	57.3	57.3	0.0%
3	7	7	65.1	66.6	2.3%
4	2	14	123.3	125.3	1.6%
5	4	16	150.2	155.9	3.8%
6	8	16	134.2	138.0	2.8%
7	5	7	92.8	97.2	4.7%
9	5	10	61.6	62.6	1.6%

Table 1. Critical path delays for the PREP benchmarks mapped to Triptych.

The results of our experiments are shown in Tables 1 and 2. Covering with three-input functions was performed with the SIS mapper. All circuits were mapped to an 8x64 array of RLBs (512 total RLBs). Table 1 shows the results of mapping the PREP benchmarks. The number of repetitions

(*Reps*) of any particular benchmark is determined by the maximum that will fit in the 8x64 array when routed using algorithm NC of Section 3.1. This insures a dense circuit and is therefore a good test to determine how well the router can optimize for delay when algorithm NCD of Section 3.2 is used.

The *Logic Levels* column in the table is the maximum number of 3-input functions between registers. *Optimal Delay* is a lower bound to the delay that can be obtained given a placement. This number is obtained during the first iteration of the router when only the delay term in the cost function is present. Note that this number may not be achievable when congestion is resolved due to competition between critical routes for the same routing resources. The *Routed Delay* column is the delay of the critical path after convergence. *% over optimal* is the % degradation of the *Routed Delay* from the *Optimal Delay*, which averages 2.1%, and is at worst 4.7%.

Benchmark	Logic Levels	Optimal Delay	Routed Delay	% over optimal
ex1	8	76.1	79.7	4.7%
keyb	10	95.4	102.9	7.8%
C880	14	153.8	159.4	3.6%
clip	11	126.0	131.0	3.9%
C1908	15	161.9	171.3	5.8%
mm9b	14	128.2	128.3	0.0%
bw	7	89.6	98.1	9.4%
s832	11	117.0	118.1	1.0%
s820	10	112.8	114.7	1.7%
x1	7	84.1	94.7	12.6%
s953	10	101.7	103.8	2.1%
s1423	30	265.2	271.0	2.1%

Table 2. Critical path delays for selected circuits from ISCAS93 mapped to Triptych.

Table 2 shows the results of running PathFinder on benchmarks obtained from ISCAS93. All circuits in the benchmark suite were included that utilized between 25% and 50% of the 8x64 array for logic (i.e. between 128 and 256 RLBs). In this case the delay degradation from optimal is an average of 4.6%, and is at worst 12.6%. The only other work quoting delay degradation from optimal is that of [Frankle92], in which an average degradation of 16% is found on the Xilinx 4000 architecture.

4.2 Experiments on Xilinx 3000

Our WireC template for the Xilinx 3000 architecture describes the entire routing structure of muxes, pips and switchboxes ([Xilinx93]). All details were specified, including routing resources to pads and clock buffers. Through parametrization of the template, we were able to describe both 3020PC68 and 3090PC84 parts.

Both CLB modes 'F' and 'FG' (corresponding to 5-input and 4-input functions) were supported; pin permutability for each mode was specified by adding additional nodes to the graph. Allowing the router to choose between input pin permutations is an important degree of freedom that often allows congestion in the neighborhood of the input pins to be resolved. Lacking detailed information describing the drive capability of individual muxes and pips, we used an additive delay model based only upon the number of fanins and fanouts of each node. In particular we did not distinguish between restoring and nonrestoring routing resources.

Tables 3 and 4 show the results for circuits taken from the ISCAS93 benchmark suite when mapped to the Xilinx 3020PC68 part and the 3090PC84 part, respectively. All circuits were fitted to Xilinx parts using the *fpga* tool from Exemplar Logic, Inc. The Xilinx tools *xnf2map*, *map2lca* and *APR* were used to map, place and route the circuits. The table column *Logic Levels* is the number of CLBs on the longest path, where a path begins at any flip-flop output or IOB, and ends at any flipflop input or IOB. The Xilinx tool *Xdelay* was used to calculate the delay of the critical path, shown in the *APR delay* column.

PathFinder reads in an unrouted *lca* file and the directed graph derived from the architecture template and performs the route. The resulting assignment of routing resources is written out in the *lca* format and checked for validity using *Xdrc*. Although PathFinder performs a delay analysis itself, the delay model is only qualitatively correct; therefore, to make a direct comparison with the Xilinx results we ran *XDelay* on our routed circuit. The results of running *XDelay* on our routed circuits is shown under the *delay* column. *% from APR* is the % difference from between this number and *APR delay.*

				PathFinder	
Bench-mark	CLBs	Logic Levels	APR delay(ns)	Delay (ns)	% over APR
s400	46	5	47.9	51.9	+8.3%
s344	32	6	56.5	55.7	-1.4%
s382	50	6	61.1	57.4	-6.1%
s386	38	5	51.6	49.3	-4.4%
s420	20	4	40.4	41.0	+1.4%
s420.1	46	4	50.2	44.4	-11.6%
s444	46	5	54.2	51.9	-4.2%
s208.1	18	4	40.6	36.6	-9.8%

Table 3. Critical path delays for selected circuits from ISCAS93 mapped to a Xilinx 3020PC68

The results from Table 3 indicate that PathFinder converges on a route with a shorter critical path than APR. The average is 3.5% shorter. It is notable that when we examine our critical paths, they often contain multiple nonrestoring pips in sequence. We would expect considerable improvement in these critical path delays given a delay model which distinguishes between restoring and nonrestoring resources.

				PathFinder	
Bench-mark	CLBs	Logic Levels	APR delay (ns)	delay (ns)	% over APR
s1	116	8	113.5(fails)	91.1	-19.7%
9sym	76	11	134.8	124.1	-7.9%
9symml	99	18	197.9	182.9	-7.6%
alu2	123	18	270.8(fails)	243.3	-10.1%
dk16	128	7	78.5	79.5	+1.3%
duke2	99	7	141.5	106.5	-24.7%
ex1	75	7	120.3	117.9	-1.9%
keyb	74	8	112.7	86.6	-23.2%
planet	151	7	194.4(fails)	179.9	-7.5%

Table 4. Critical path delays for selected circuits from ISCAS93 mapped to a Xilinx 3090PC84.

Of the nine 3090 circuits shown in Table 4, APR was unsuccessful at routing three of them -- s1 (20 unrouted pins), alu2 (12 unrouted pins), and planet (29 unrouted pins). PathFinder was successful with all three circuits, suggesting that our iterative approach to alleviating congestion is an improvement over the standard obstacle avoidance methods, such as that employed by APR. An average of 11.3% improvement over the APR delays was obtained on the circuits in Table 4. This is significantly greater than that obtained for the 3020 circuits in Table 3 because the 3090 circuits have longer routes on average, increasing the proportion of routing delay in the total delay, which is the sum of logic and routing delays. It should be noted that the percent improvement in the routing delay is much larger than 11.3%, since the critical path delay includes logic which is not affected by the router.

5 Conclusions and Future Work

We have presented PathFinder, a new, iterative routing algorithm for FPGAs. Results on the Triptych FPGA architecture show only a small degradation of critical path delay from the optimum for a given placement, indicating that the algorithm is minimizing congestion while meeting delay constraints. Results on the Xilinx 3000 architecture show a higher degree of routability as well as 11% shorter critical paths than commercial tools are able to achieve. This is especially encouraging given the lack of detailed delay information on the routing resources in the Xilinx 3000 architecture.

Our conclusion from this work is that a carefully-designed iterative approach to routing on a directed graph allows one to optimize over multiple and, in some cases, competing objectives. In the case of FPGAs this approach is successful at optimizing delay while eliminating congestion.

Acknowledgments

We want to thank Gaetano Borriello for helpful discussions about routing for Triptych and Steven Yee for his considerable help in constructing the Xilinx3000 template. The authors also acknowledge Darren Cronquist and Scott Hauck for suggestions to improve this manuscript. This work was funded by the Advanced Research Projects Agency under contract N00014-J-91-4041.

References

[Alexander94] M. Alexander, "A Unified New Approach to FPGA Routing Based on Multi-Weighted Graphs," *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, February 1994.

[Brown92] S. Brown, J. Rose, and Z. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 5, May 1992, pp. 620-628.

[Cohn91] J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, "KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing," *IEEE Journal of Solid-State Circuits*, vol. 26, March 1991, pp. 330-342.

[Dees81] W. Dees and R. Smith, "Performance of Interconnection Rip-Up and Reroute Strategies," in *Proc. 18th Design Automation Conference*, June 1981, pp. 382-390.

[Frankle92] J. Frankle, "Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing," in *Proc. 29h Design Automation Conference*, June 1992, pp. 536-542.

[Hauck92] S. Hauck, G. Borriello and C. Ebeling, "TRIPTYCH: An FPGA Architecture with Integrated Logic and Routing," in *Proc. of the 1992 Conference on Advanced Research in VLSI and Parallel Systems*, March 1992, pp. 26-43.

[Hill91] D. Hill, "A CAD System for the Design of Field Programmable Gate Arrays," in *Proc. 28th Design Automation Conference*, June 1991, pp. 187-192.

[Linsker84] R. Linsker, "An Iterative-Improvement Penalty-Function-Driven Wire Routing System," *IBM Journal of Research and Development*, vol. 28, Sept. 1984, pp. 613-624.

[Nair87] R. Nair, "A Simple Yet Effective Technique for Global Wiring," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, March 1987, pp. 165-172.

[Takahashi80] H. Takahashi and A. Matsuyama, "An Approximate Solution for the Problem in Graphs," *Japonica*, vol. 24, 1980, pp. 573-577.

[Xilinx93] Xilinx, Inc., *Xact Development System*, 1993.