QuickRoute: A Fast Routing Algorithm for Pipelined Architectures[†]

Song Li[‡] and Carl Ebeling
Department of Computer Science and Engineering
University of Washington
Seattle, WA, USA
{songli,ebeling}@cs.washington.edu

Abstract

As interconnect delays begin to dominate logic delays in large circuits, pipelined interconnects will be needed to achieve the highest performance. In FPGAs, this pipelining will be provided by the configurable interconnect architecture itself. This changes the routing problem substantially since the shortest path problem, which is at the core of any router, becomes NP-hard when latency constraints are added. That is, if signals must be routed through a given number of registers between source and destination, an efficient shortest path algorithm like Djikstra's algorithm is no longer an option. We propose here an approximate algorithm that uses simple heuristics to solve the pipelined shortest path problem efficiently. We have incorporated QuickRoute in the PathFinder router to route pipelined interconnects. We present the results achieved with QuickRoute for several circuits with heavily pipelined interconnect which show an improvement over a previously described algorithm.

1. Introduction

The pipelined routing problem is becoming increasingly important for FPGA architectures. As FPGAs become larger and the interconnect delay for long signals constrains performance, there is a need to support pipelined signals. Pipelined interconnect architectures have already been proposed for FPGAs to address this problem [1-4], but even existing FPGAs have an excess of registers that can be used for pipelining long signals. Although it is possible to place signal pipeline registers during placement and before routing, it is much more effective to do so during routing. In this case, the routing algorithm must not only find a path from source to sink, but find a path that includes a given number of registers. Unfortunately, this changes the very easy shortest path problem into an NP-hard problem [5, 6].

A second context for pipeline routing occurs when mapping algorithms to coarse-grained reconfigurable architectures [1, 7, 8]. Running an application on a coarse-grained pipelined FPGA architecture is comparable to running the application on a VLIW processor where the application runs as a loop of several cycles and the hardware is time-multiplexed. Mapping to one of these architectures requires solving the scheduling problem while additionally solving the placement and routing problem. The scheduling problem can be addressed using an Iterative Modulo Scheduling algorithm [9, 10], which schedules the operations in a loop iteration using the resources for a fixed number of clock cycles called the iteration interval (II). After the operations have been scheduled in time, they must be placed and routed onto the architecture to meet the constraints of the interconnect. This routing problem is a pipelined routing problem because values must be forwarded in general across a given number clock cycles, which means finding paths with a given number of registers.

These two problems are in fact very similar. The pipelined FPGA routing problem is actually the IMS routing problem with II = 1. That is, although a pipelined FPGA does not time-multiplex the hardware, values are forwarded from one function unit to another with potentially several clock cycles of latency. In this paper, we describe the QuickRoute algorithm for II = 1, that is, for the FPGA pipelined routing problem, to simplify the presentation. The algorithm can be extended to the more general pipelined routing problem, and in fact, it is more appropriate for routing coarse-grained configurable architectures. The case where II = 1 presents the most challenging problem, and thus we have chosen to validate it for this case.

We have incorporated the QuickRoute shortest-pipelined path-algorithm into the PathFinder algorithm [11-13], where it replaces Dijkstra's shortest path algorithm. PathFinder is a negotiation-based algorithm widely used for FPGAs that iteratively solves the congestion problem. The inner-loop of PathFinder relies on Dijkstra's shortest path algorithm to route individual source-sink pairs. Fanout is typically handled by routing to the closest sink first, and then to the remaining sinks in order, starting from the already established signal

[†] This material is based upon work supported by the National Science Foundation under Grant CCR-0220095.
[‡] Song Li is now with Altera Corporation in San Jose, Ca.

routing tree. The A* algorithm [14, 15] is generally used to direct the search in the direction of the sink: An estimate of the remaining distance to the sink is used to choose which node to visit next when extending the search. Using A* can greatly increase the speed of the PathFinder search [16].

Since the shortest path algorithm is used in the inner loop of the router, it must be very fast. It is not crucial, however, for it to be optimal. That is, the overall quality of the router results does not degrade greatly if the algorithm sometimes finds a path that is slightly longer than the actual shortest path. Thus, the fact that QuickRoute necessarily cannot guarantee shortest paths is not fatal as long as the approximate solutions it finds are close to optimal.

While QuickRoute can fail in the worst case, we argue that this does not occur for reasonable architectures and reasonable routing problems.

2. Related Work

There has been very little research reported on the shortest pipelined path problem. Variants of this problem appear in the literature as special cases of the traveling salesman problem known sometimes as the "quota traveling salesman" or the "prize-collecting traveling salesman" problem [6]. The important fact is that these problems are NP-complete and thus approximation algorithms must be used for large problems. The theoretical literature focuses on good approximation algorithms that run in polynomial time but these are not practical for the inner loop of a router that must deal with very large graphs.

PipeRoute [17, 18] solves the pipelined shortest path problem using a divide-and-conquer algorithm. First, an efficient algorithm is described for routing a signal containing either 0 or 1 registers. The general n-register pipelined routing problem is then solved recursively using multiple 1-register routes. We compare QuickRoute to PipeRoute using the same benchmark circuits.

3. The Shortest Pipelined Path Problem

The shortest pipelined path problem is defined as follows: Given a source node and a sink node in a directed circuit graph and $N \geq 0$, find the shortest path from source to sink that includes exactly N registers. This problem, unfortunately, is NP-hard as previously mentioned, and so we must use approximate algorithms. We will begin by describing a brute-force algorithm that is a generalization of Dijkstra's shortest-path algorithm. This algorithm solves the shortest pipelined path problem, albeit in exponential time. We will then add heuristic pruning to this algorithm to define the QuickRoute algorithm. The result is that QuickRoute is only approximate; however, we show experimentally that it works very well in practice for realistic architectures.

Figure 1 shows Dijkstra's algorithm. In this formulation, a path is a sequence of nodes that is maintained implicitly using backpointers in the graph data structure. The cost of a path is the sum of the cost of the nodes in the path. The algorithm works by extending the current shortest (cheapest) partial path until a path reaches the sink. Since this algorithm explores the shortest paths first, the first path to reach a node must be the shortest path to that node. Thus, any shortest path to the destination through a node can use the first shortest path from the source to the node as a prefix. Thus each node needs to be visited at most once, and this is implemented by marking nodes.

```
Dijkstra (source, sink)
  1   Unmark all nodes
  2   Initialize the priority queue Q
  3   Insert the path [source] into Q
  4   While Q is not empty do
  5     Remove the shortest path P from Q
  6     if P.end == sink then return P
  7     else
  8       for every neighbor n of P.end do
  9         if n is not marked then
  10          mark n as visited
  11          add new path P' = [P, n] to Q
                with cost = P.cost + n.cost
  12        endIf
  13      endFor
  14    endIf
  15  endWhile
  16  return failed
```

Figure 1 - Dijkstra's algorithm

Of course, Dijkstra's shortest path algorithm does not work for the shortest pipelined path problem because, for $N>0$, the first path to a node may not form the prefix of any path to the destination node that meets the latency constraint. And even if it does, there may be a different shorter path through the node that meets the latency constraint.

An alternative to Dijkstra's algorithm is to search all possible paths, and find the shortest one with the given latency. This brute force algorithm, expressed as an extension of Dijkstra's algorithm is shown in Figure 2. In this algorithm, the latency of a path refers to the number of registers on the path.

The brute force algorithm is different from Dijkstra's algorithm in one very important respect: it does not mark nodes when they are included in a path and therefore does not prune paths. This means that **all** paths from source to destination are examined in order of cost. In order to avoid using a node twice on a path, the algorithm must check when extending a path that the path does not already include the node. This is a check that Dijkstra's algorithm does not need since nodes can only be visited once. The brute force search guarantees it will find the shortest path with the required latency.

However, the brute force algorithm searches all possible paths shorter than the shortest path and if there are N nodes in the graph, then there may be as many as 2^N paths in this search.

The QuickRoute algorithm shown in Figure 3 uses a simple heuristic to efficiently prune the search. Instead of searching all possible paths, once k paths to a node n has been found with latency l, then all other paths through n with latency l are pruned. That is, QuickRoute keeps a count of how many paths have visited a node for each latency. Only the first k paths to visit a node at a given latency are kept for further expansion – all other paths are pruned. The resulting search is more expensive than the simple shortest path algorithm, but only by a constant factor. The k parameter can be adjusted to make the search more or less broad, but we have found that k=1 is sufficient in practice to yield good results.

```
BFSearch (source, sink, N)
initialize the priority queue Q
insert the path [source] into Q
while Q is not empty do
   remove the shortest path P from Q
   if P.end == sink and P.latency == N then
      return P
   else
      for every neighbor n of P.end do
         if n is not in P then
            add new path P' = [P, n] to Q with
                  cost = P.cost + n.cost
         endIf
      endFor
   endIf
endWhile
return failed
```

Figure 2 - Brute force pipelined search algorithm

```
QuickRoute (source, sink, N)
for all nodes n, for all i<=N, n.visited[i] = 0.
initialize the priority queue PQ
insert the path [source] into PQ
while PQ is not empty do
   remove the shortest path P from PQ
   if P.end == sink and P.latency == N then
      return P
   else
      for every neighbor n of P.end do
         if n.visited[P.latency] < k and
                  n is not in P then
            add new path P' = [P, n] to PQ with
                  cost = P.cost + n.cost
            increment n.visited[P.latency]
         endIf
      endFor
   endIf
endWhile
return failed
```

Figure 3 - QuickRoute algorithm

It is possible to construct architectures and circuit examples that cause the heuristic to fail to find any path even if there is one. One such example is shown in Figure 4. In this example, QuickRoute prunes a path to an intermediate node that is longer than a previous path to the node with same latency, but where the shorter path is not a valid prefix of any path to the destination. This can happen when paths block themselves, as shown in this example.

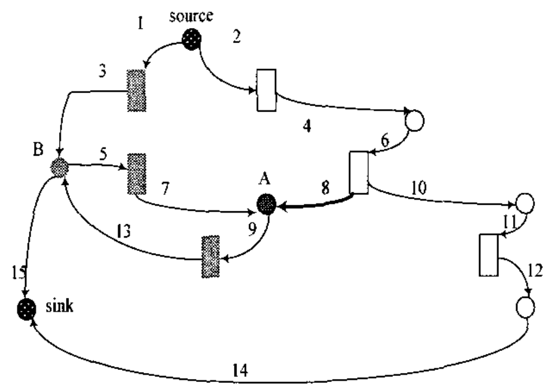

Figure 4 – An example where QuickRoute cannot find a shortest path

In the figure, the number on each edge gives the order in which QuickRoute algorithm expands the search branches. Note that branch 7 visits routing resource A first, and branch 8 cannot expand to this resource, because its latency is the same as branch 7. However, further expansion of branch 7 cannot find a valid path. Branch 13 cannot expand to the resource B since it was already visited by branch 3, which is an ancestor of branch 13. The path found by QuickRoute is 2-4-6-10-11-12-14, which is much longer than path 2-4-6-8-9-13-15. If the architecture does not allow branch 14, QuickRoute will not be able to find any path.

While the fact that QuickRoute is not guaranteed to find a path may seem fatal, our experiments show that it does in fact find paths for routes that appear in practice. In this example, the routing architecture is severely constrained. As our experiments show, the QuickRoute heuristic performs well even on difficult routing problems, although for longer latencies, only approximations to the shortest path may be found.

4. Implementation Details

The shortest path algorithm is used in the inner loop of any router and so the algorithm's performance is crucial to the performance of the router. In this section, we will describe the implementation details that are used to make QuickRoute as efficient as possible.

There are three operations in the innermost loop of QuickRoute that are potentially expensive in space or time:

1) Checking and incrementing a node's "visited count".

2) Checking whether a node n has already been used as a part of path P.
3) Adding the new path to the priority queue and removing the path with lowest cost from the queue.

With the visited limit k of 1, each visited count can be implemented as a single bit, and the vector of visited counts for each node is a simple bit vector. Since latencies do not exceed 64 for real problems, only 8 bytes per node are required for the visited counts. Checking and setting these counts are very fast, constant-time, bit-vector operations.

Checking if a node is part of a path is done by assigning each node a unique ID and keeping a vector of the IDs of the nodes in each path. This vector of IDs is a bit vector, R, where R[ID]=1 if node ID is in the path. The size of the bit vector must be as large as the largest ID. This bit-vector size is minimized using two techniques. First, each search reassigns node IDs starting from 0 as nodes are encountered in the search. Thus the size of the bit vector need only accommodate the number of nodes encountered in a search, not all the nodes in the graph. Second, only multiplexor nodes are counted, which reduces the size of the bit vector by about a factor of 3. This works because in our graph representation, each branch in the search goes through a different multiplexor. It is easy to show that if two paths share a node, then they must also share a multiplexor, and thus it is sufficient to only keep track of multiplexors. Even with these optimizations, this bit vector can become large for large architectures. However, for the coarse-grained architectures that we are targeting, the number of nodes is on the order of a few thousand, which requires on the order of 100 bytes.

The performance of the priority queue is also critical to the algorithm performance. We implemented the priority queue as a heap, using buckets for all paths with the same cost. Thus, the average insertion and access time the priority queue is logB, where B is the number of different path costs for paths in the queue.

To analyze the time complexity of QuickRoute, let N be the number of resources in the circuit graph. The algorithm may visit each resource L times, where L is the latency of the signal being routed. Thus, in the worst case, QuickRoute will execute the loop at line 4 in Figure 3 N*L times. Each execution involves a delete and insertion on the priority queue requiring logB time plus checking and setting marks, which takes constant time. Since the fanout for realistic architectures is constant, the loop at line 8 takes constant time. Thus the time complexity of QuickRoute is O(NLlogB). It is more instructive to compare the running time of QuickRoute to that of Dijkstra's algorithm. The main difference is that QuickRoute visits nodes as many as L times. This means that QuickRoute takes approximately L' times as long as Dijkstra's algorithm, where L' is the average number of times each node is visited.

The space complexity of QuickRoute limits its scalability more than the time complexity. The history bit vector grows as the number of nodes visited times the signal latency. In the general case, the space complexity is $O(N^2L)$. In practice, even with the optimizations we have described, this limits the size of the circuit the graph to about 10,000 resources, which is sufficient for coarse-grained architectures like RaPiD. Reducing the space complexity of QuickRoute is a topic for further research.

5. Experimental Results and Performance Analysis

In this section we present three experiments we conducted on the QuickRoute algorithm. The first experiment tests the ability of QuickRoute to find paths for very difficult routes with very high latencies. The second tests how close QuickRoute comes to finding the shortest pipelined path. The final experiment compares the results of using PathFinder/QuickRoute to previously published results for PipeRoute [5].

The first experiment uses QuickRoute to find the shortest path for all source-destination pairs in an instance of the RaPiD architecture [1]. The purpose of this experiment is to determine whether the QuickRoute algorithm can find paths with very difficult latency constraints. The experiments were done using the single-cell RaPiD datapath graph shown in Figure 5. This RaPiD cell has two input streams on the left side and two output streams on the right side. There are 4 functional units, two adders and two multipliers. Each functional unit has 3 datapath registers directly connected to its output which can only be used to pipeline outputs of that unit. There are a total of 81 registers: 39 registers in the bus connectors, 12 registers associated with function units, 18 registers in datapath registers, and 12 registers in the datapath memories. In the RaPiD architecture, bus connectors are used to connect segments in the same track and a signal can be pipelined as it passes from one segment to the next. However, changing tracks requires going through a datapath register. Highly pipelined signals must be routed using all available registers, including all datapath registers, memory registers and as many bus connector registers as possible. With the constraints on the routing architecture, the maximum latency of any one signal is less than 60.

In this experiment, all source-destination pairs were routed for latencies from 0 to 50. The most difficult routes are those from a function unit output back to its input with a large latency. Instead of finding the "easy" short path from output back to the input, the algorithm must "snake" a route through the entire cell, finding as many registers as possible, switching tracks via datapath registers to visit extra bus connectors, before returning to the same function unit. Readers familiar with Dijkstra's algorithm will realize that a naïve algorithm will generate outgoing paths that block any chance for the path to return.

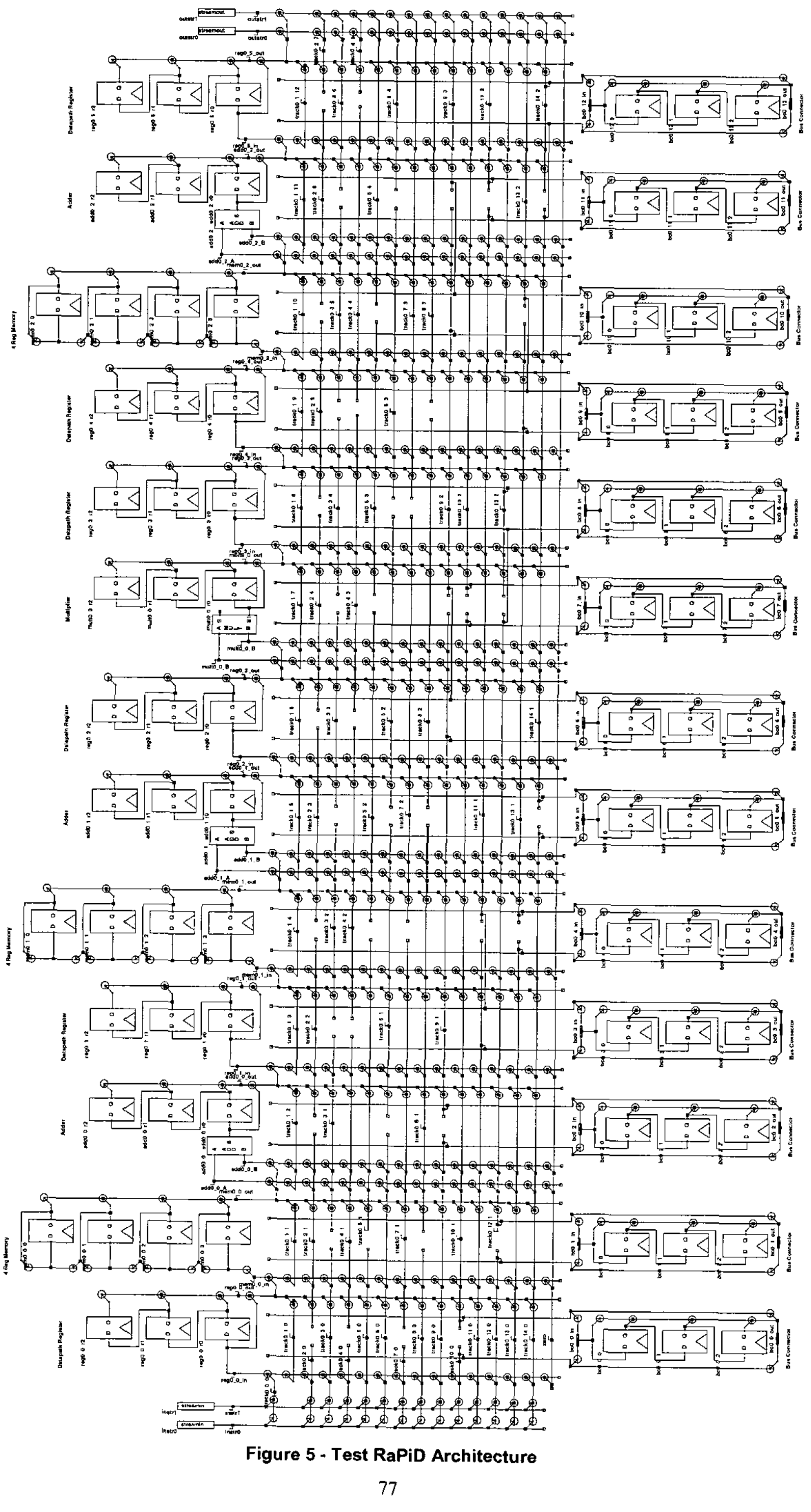

Figure 5 - Test RaPiD Architecture

In this experiment, QuickRoute was able to find paths for all pairs and all latencies through 42. It was also able to find many paths for higher latencies as well. Note that a latency of 42 can be attained only by using all the originating functional unit output registers, all the datapath registers, all the memory registers and all the registers in three bus connectors. Using more than 42 registers requires switching tracks, which means using datapath registers to both switch tracks and pipeline the signal. This is a very difficult solution to find and yet QuickRoute is able to find it for many long routes.Figure 6 shows the shortest path found by the QuickRoute algorithm from the ALU output back to an input of the same ALU with a path latency of 8. The ALU is highlighted as the black rectangle in the middle of the graph. The registers and nets of the shortest path are highlighted in green. The path starts from the output pin of the ALU, goes through the nearest 8 registers and feeds in the first input pin of the ALU. It is easy to determine that this is in fact a shortest path.

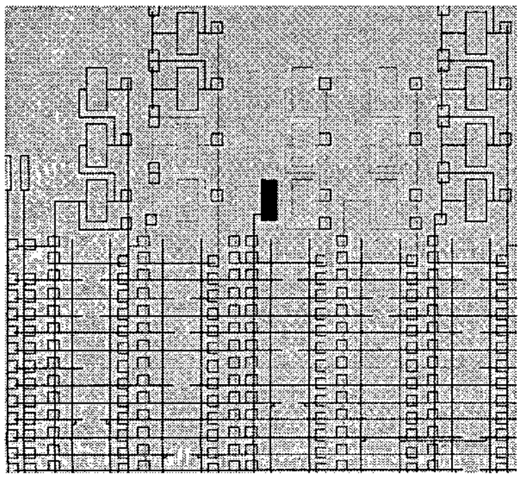

Figure 6 - An example of the shortest path found by the QuickRoute algorithm. Latency is 8.

The subset of the results of these experiments is shown in Figure 7. The numbers show the cost of the shortest path from add0 to each of the sinks for the latencies from 0 to 49. An entry of -1 shows the cases where QuickRoute failed to find a route. Note that the most difficult case occurs when a signal is routed from add0 back to add0. Note also that the out0 and out1 sinks are the farthest from add0 for 0 latency. But for larger latencies, it is cheaper to route to these than it is to route a signal back close to the source. For very large latencies, all of the routes must snake through most of the available registers and end up with about the same cost.

The goal of the second experiment was to understand how good an approximation to the shortest pipelined path QuickRoute is able to find. It is not feasible to write a program to compute the shortest path for reasonably-sized circuits since the problem is NP-hard. Instead, we manually determined the shortest path costs for many different source-destination pairs with latencies up to 16 registers using our intimate knowledge of the architecture. Although it may appear to be difficult to find these shortest paths, in most cases it easy to determine the cost of the shortest path with latency i+1 using the cost of the shortest path with latency i. QuickRoute found a minimum-cost path for all these routes.

For paths with longer latencies, manually finding the shortest path was not feasible and so we used an alternative approach to determine whether the QuickRoute algorithm had found the shortest path. In the QuickRoute algorithm, Figure 7, line 5, the search can choose to visit any of the current shortest paths next, and this can affect the result of the search. In the example of Figure 4, the 7th search branch that visits a resource cannot reach the destination resource as the shortest path. If during the search, the 8th branch reaches the resource before the 7th branch does, the QuickRoute algorithm will be able to find the shortest path.

We modified QuickRoute to pick the next path randomly from the head of the priority queue. We then re-ran the all-pairs experiment above for latencies from 16 to 25, running QuickRoute 256 times for each source/sink pair, each time with a different random number seed. We wanted to see if QuickRoute would always find a shortest path or whether was some variation due to the random choice. For latencies between 15 and 22, the QuickRoute algorithm always found the shortest path. However, for latencies larger than 22, there was a variation of up to 10% in the cost of the paths found. That is, spending more effort by randomly exploring more paths can find a shorter path when latency is greater than 22 by as much as 10%. These results show that the QuickRoute algorithm is near optimal for realistic latencies, becomes increasing approximate for larger latencies and finally fails for very large latencies that strain the register capacity of the interconnect.

The third experiment compares PathFinder using QuickRoute for routing complete circuits to an existing routing algorithm called PipeRoute [5] which also targets pipelined interconnect architectures. We used the same application netlists and architectures used for the results reported in this earlier paper. We first converted the circuit netlists into scheduled dataflow graphs, which our PathFinder/QuickRoute tool uses. We also used an identical placement of the circuit netlists onto identical versions of the RaPiD architecture. We had to constrain the PathFinder/QuickRoute tool to not use function output registers and datapath registers since PipeRoute could not use them. (PipeRoute has since been modified to be able to use all registers.) The experiment was run in the standard way to determine the minimum number of tracks needed to completely route the circuit. The router was repeatedly used to route a circuit to the architecture with different numbers of tracks until a successful route was achieved.

k	out0	out1	add0	add1	add2	mult
0	42	42	10	14	38	22
1	43	43	12	16	38	24
2	45	45	14	18	40	26
3	47	47	16	20	42	28
4	49	49	26	24	44	30
5	51	51	28	26	46	32
6	53	53	30	28	48	34
7	55	55	40	31	50	36
8	57	57	44	35	53	38
9	59	59	48	39	55	40
10	61	61	52	43	57	46
11	64	64	60	57	60	50
12	66	66	64	61	62	54
13	69	69	68	65	65	58
14	72	72	78	75	68	72
15	74	74	82	82	70	76
16	77	77	86	88	73	80
17	80	80	90	92	77	90
18	82	82	101	96	81	96
19	88	88	105	100	85	100
20	92	92	109	110	91	108
21	96	96	116	114	95	112
22	100	100	120	118	99	116
23	106	106	124	122	103	120
24	110	110	133	131	117	130
25	114	114	137	135	121	134
26	118	118	143	139	125	139
27	138	138	147	146	135	144
28	142	142	151	150	144	148
29	146	146	155	154	148	152
30	150	150	164	165	155	165
31	162	162	168	169	169	169
32	166	166	172	173	173	173
33	170	170	177	186	177	184
34	214	214	184	193	192	196
35	227	227	188	197	196	200
36	231	231	192	201	200	204
37	226	226	201	210	226	217
38	230	230	205	214	230	221
39	234	234	209	218	234	225
40	236	236	216	225	243	233
41	240	240	220	229	247	237
42	244	244	224	233	251	241
43	258	258	-1	258	258	258
44	262	259	-1	262	262	262
45	266	266	-1	266	266	266
46	277	277	-1	-1	277	-1
47	281	281	-1	-1	281	-1
48	285	285	-1	-1	285	-1
49	-1	-1	-1	-1	-1	-1

Figure 7 - Routing results for selected source-destination pairs and latencies 1-49

Applications	PipeRoute	QuickRoute
cascade	9	9
firsymeven	18	15
firtm	18	21
mat_mult4	12	8
sortG	20	20
sort_2D_RB	16	11
med_filt	20	17

Figure 8 - Comparing the relative routing results for PipeRoute and QuickRoute.

Figure 8 shows the number of tracks used to route each application using QuickRoute and PipeRoute. The QuickRoute Algorithm uses same or fewer tracks than the PipeRoute algorithm in 6 out of the 7 applications. It uses fewer tracks than the PipeRoute algorithm in 4 applications: firsymeven, mat_mult4, sortG and sort_2D_RB, it uses the same number of tracks in cascade and med_filt. The only application for which the QuickRoute algorithm uses more tracks than the PipeRoute algorithm is firtm, where the QuickRoute algorithm uses 3 more tracks than the PipeRoute algorithm.

Applications	Time(sec)
cascade	12.1
firsymeven	26.4
firtm	18.2
mat_mult4	16.2
sortG	14.1
sort_2D_RB	18.2
med_filt	65.6

Figure 9 - The running time of the QuickRoute algorithm

Since the QuickRoute algorithm is implemented using Java and the PipeRoute algorithm is implemented using C++, we cannot make a direct comparison of the running time of the two algorithms. Figure 9 shows the running time of the QuickRoute algorithm for the different applications. The experiments are done on a Pentium IV machine with a clock rate of 3.0GHz and 2.0GB memory. These results support our assertion that QuickRoute is not limited by time complexity, but rather by space complexity.

6. Conclusion and Future Work

In this paper, we have presented a new algorithm for the shortest pipelined path problem that is both efficient and effective. Our experiments show that it is an improvement over previous attempts to solve this problem. Although the algorithm in its current form does not scale to extremely large circuits, it is appropriate for the coarse-grained configurable architecture scheduling problem that we are currently working on. In this problem, dataflow graphs are scheduled over several clock cycles, which complicates the algorithm somewhat. However, the basic algorithm extends easily to this problem and preliminary experimental results are very encouraging.

Our research is now focused in two directions. First, we are working on improving the QuickRoute algorithm to make it applicable to a wider range of circuits. We are working on techniques that will make the QuickRoute algorithm space-efficient without significantly impacting the quality of the results. We are experimenting with different ways to use A* in the contest of pipelined routing. It is clear that using distance estimates in the pipelined routing problem, especially where II ≠ 1, is very tricky. The role of multiplexors in the reconfigurable datapath also needs to be addressed. Multiplexors used for predicated execution can either be placed by the placement tools or by the router. Currently we use the placement tools for placing multiplexors, but preliminary results show that the routing algorithm can generate better placements.

Second, we are using QuickRoute as a part of a larger set of tools for scheduling, placement and routing of large dataflow graphs to coarse-grained configurable architectures. The goal of this work is to produce a compilation system that is architecture-independent within a fairly general model of coarse-grained configurable architectures. QuickRoute will be a crucial part of this system.

Acknowledgments

We would like to express our appreciation to Chris Fisher, who provided the visualization code, Akshay Sharma who generously helped us set up the experiments for the PipeRoute comparisons, and Scott Hauck for many helpful comments and suggestions.

References

[1] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture design of reconfigurable pipelined datapaths," The Conference on Advanced Research in VLSI, Atlanta, 1999.

[2] A. Singh, A. Mukherjee, and M. Marek-Sadowska, "Interconnect pipelining in a throughput-intensive FPGA architecture," Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays, 2001.

[3] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon, "HSRA: high-speed, hierarchical synchronous reconfigurable array," Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays, 1999.

[4] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: a Coprocessor for Streaming Multimedia Acceleration," 26th International Symposium on Computer Architecture (ISCA99), 1999.

[5] A. Sharma, C. Ebeling, and S. Hauck, "PipeRoute: A PipeliningAware Router for FPGAs," University of Washington, EE Department Technical Report UWEETR-0018, 2002.

[6] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala, "Improved approximation guarantees for minimum-weight k-trees and prize-collecting salesmen," Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, Las Vegas, Nevada, United States, 1995.

[7] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," IEEE Computer, vol. 33, 2000.

[8] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," Computers, IEEE Transactions on, vol. 49, pp. 465-481, 2000.

[9] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," Proceedings of the 27th annual international symposium on Microarchitecture, 1994.

[10]B. Rau, "Iterative Modulo Scheduling," HP Labs Technical Report HPL-94-115, 1994.

[11]L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," Proceedings of the third international ACM symposium on Field-programmable gate arrays, 1995.

[12]C. Ebeling, L. McMurchie, S. A. Hauck, and S. Burns, "Placement and routing tools for the Triptych FPGA," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 3, pp. 473-482, 1995.

[13]V. Betz, J. Rose, and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs: Kluwer Academic Publishers, 1999.

[14]N. J. Nilsson, Principles of artificial intelligence: Morgan Kaufmann Publishers Inc., 1980.

[15]R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality af A*," J. ACM, vol. 32, pp. 505--536, 1985.

[16]J. S. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Monterey, California, United States, 1998.

[17]A. Sharma, C. Ebeling, and S. Hauck, "PipeRoute: A pipelining-aware router for FPGAs," ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays (FPGA03), 2003.

[18]A. Sharma, K. Compton, C. Ebeling, and S. Hauck, "Exploration of Pipelined FPGA Interconnect Structures," Twefth International Symposium on Field-Programmable Gate Arrays, Monterey, CA, 2004.