PipeRoute: A Pipelining-Aware Router for Reconfigurable Architectures

Akshay Sharma, *Student Member*, *IEEE*, Carl Ebeling, *Member*, *IEEE* and Scott Hauck, *Senior Member, IEEE*

***Abstract*—We present a pipelining-aware router for FPGAs. The problem of routing pipelined signals is different from the conventional FPGA routing problem. The two terminal N_D pipelined routing problem is to find the lowest cost route between a source and sink that goes through at least N ($N \geq 1$) distinct pipelining resources. In the case of a multi-terminal pipelined signal, the problem is to find a Minimum Spanning Tree that contains sufficient pipelining resources such that pipelining constraints at *each* sink are satisfied. In this work, we first present an optimal algorithm for finding a lowest cost 1_D route. The optimal 1_D algorithm is then used as a building block for a greedy two terminal N_D router. Next, we discuss the development of a multi-terminal routing algorithm (PipeRoute) that effectively leverages both the 1_D and N_D routers. Finally, we present a pre-processing heuristic that enables the application of PipeRoute to pipelined FPGA architectures. PipeRoute's performance is evaluated by routing a set of benchmark netlists on the RaPiD architecture. Our results show that the architecture overhead incurred in routing netlists on RaPiD is less than 20%. Further, the results indicate a possible trend between the architecture overhead and the percentage of pipelined signals in a netlist.**

***Index Terms*—Design automation, Field programmable gate arrays, Reconfigurable architectures, Routing**

I. Introduction

Over the last few years, reconfigurable technologies have made remarkable progress. Today, state-of-the-art devices [1], [19] from FPGA vendors provide a wide range of functionalities. Coupled with gate-counts in the millions, these devices can be used to implement entire systems at a time. However, improvements in FPGA clock cycle times have consistently lagged behind advances in device functionality and capacities. Even the simplest circuits cannot be clocked at more than a few hundred megahertz.

Manuscript received February 7, 2005. This work was supported by grants from the National Science Foundation (NSF). Scott Hauck was supported in part by an NSF Career Award and an Alfred P Sloan Fellowship.

Akshay Sharma is with the Electrical Engineering Department, University of Washington, Seattle WA 98195 USA (email akshay@ee.washington.edu).

Carl Ebeling is with the Computer Science and Engineering Department, University of Washington, Seattle WA 98195 USA (email ebeling@cs.washington.edu).

Scott Hauck is with the Electrical Engineering Department, University of Washington, Seattle WA 98195 USA (email hauck@ee.washington.edu).

A number of research groups have tried to improve clock cycle times by proposing pipelined FPGA architectures. Some examples of pipelined architectures are HSRA [16], RaPiD [3] – [4], and the architecture proposed in [13]. The main distinguishing feature of a pipelined FPGA is the possible location of registers in the architecture. While both pipelined and conventional FPGA architectures provide registers in the logic structure, pipelined FPGAs also provide register sites in the interconnect structure. The registers in the interconnect structure are used to supplement the registers in the logic structure. Applications mapped to pipelined FPGAs are often retimed to take advantage of a relatively large number of registers in the logic and interconnect structures.

The subject of this paper is the development of an algorithm called PipeRoute that routes application netlists on pipelined FPGA architectures. PipeRoute takes a netlist and a pipelined FPGA architecture as inputs, and produces an assignment of signals to routing resources as the output. To the best of our knowledge, PipeRoute is the first algorithm that is capable of routing netlists on pipelined FPGA architectures. Furthermore, the strength of the PipeRoute algorithm lies in the fact that it is architecture-independent. The algorithm is capable of routing pipelined signals on any FPGA architecture that can be abstracted as a graph consisting of routing and pipelining nodes.

The rest of this paper is organized as follows. In Section II, we survey pipelined FPGAs and the techniques used to allocate pipelining registers during the physical design phase. The pipelined routing problem is introduced and formalized in Section III. In sections IV, V and VI we present PipeRoute [10] – [12], a greedy heuristic search algorithm for routing signals on pipelined FPGA architectures. Section VII discusses a pre-processing heuristic that takes advantage of registered IO terminals and interconnect sites that provide *multiple* registers. In Section VIII, we briefly describe the techniques we used to make PipeRoute timing aware. The target architecture that we used in our experiments is described in Section IX, while Section X explains the placement algorithm we developed to enable our routing approach. We describe our experimental setup and test strategy in Section XI, followed by results in Section XII. In Section XIII, we conclude this paper.

II. Pipelined Fpgas And Mapping Heristics

In this section we briefly survey examples of pipelined

FPGA architectures, and the heuristics used to allocate pipelining registers during physical design.

A. Fixed-frequency Architectures

HSRA [16] and SFRA [17] are two examples of *fixed-frequency* architectures that guarantee the execution of an application at a fixed frequency. HSRA has a strictly hierarchical, tree-like routing structure, while SFRA has a capacity depopulated island style routing structure. Applications mapped to HSRA and SFRA are aggressively C-slowed to reduce clock cycle times. An important consequence of C-slowing is that the register count in a netlist increases by a factor of C. Since an FPGA has limited resources, finding pipelining registers in the interconnect and/or logic structure without adversely affecting the routability and delay of a netlist is a difficult problem. Both HSRA and SFRA circumvent this problem by providing deep retiming register-banks at the inputs of logic blocks, as well as registered switch-points.

Since fixed-frequency FPGAs provide register-rich logic and routing structures, there is no need to efficiently locate pipelining registers during placement and routing. Consequently, the place & route flows developed for SFRA and HSRA are unaware of pipelining registers. However, the area overhead incurred by these architectures due to their heavily pipelined structure is high. HSRA incurs approximately a 2X area overhead, while SFRA takes a 4X area hit. While these overheads might be justifiable for certain classes of applications (those that are amenable to C-slowing and/or heavy pipelining), they are prohibitive for conventional, general-purpose FPGAs.

B. General-purpose FPGA Architectures

A number of researchers have attempted to integrate pipelining with place and route flows for general-purpose FPGAs. The techniques presented in [8] use post place-and-route delay information to accurately retime netlists mapped to the Virtex™ family of FPGAs. To preserve the accuracy of delay information, the authors do not attempt to re-place-and-route the netlist after the completion of the retiming operation. Since the logic blocks (called 'slices') in Virtex™ devices have a single output register, the edges in the retiming graph are constrained to allow no more than a single register. This constraint might be overly restrictive, especially for applications that might benefit from a more aggressive retiming approach like C-slowing.

In [18], a post-placement C-slow retiming technique for the Virtex™ family of FPGAs is presented. Since C-slowing increases the register count of a netlist by a factor of C, a post-retiming heuristic is used to place registers in unused logic blocks. A search for unused logic blocks is begun at the center of the bounding box of a net. The search continues to spiral outward until an unused register location is found. When an unused register location is found, it is allocated. This process is repeated until all retiming registers have been placed. A significant shortcoming of this heuristic is its dependence on the pre-retiming placement of a netlist. If the placement of the netlist is dense, then the heuristic may not be able to find unused register locations within the net's bounding box. Instead, unused locations that are far removed from the net's terminals may be allocated. The resultant routes between the net's terminals and the newly allocated registers might become critical and thus destroy the benefits of C-slowing.

An alternative approach to locate post-placement retiming registers is presented in [15]. The newly created registers are initially placed into preferred logic blocks even if the logic blocks are occupied. A greedy iterative improvement technique then tries to resolve illegal overlaps by moving non-critical logic blocks to make space for registers. The cost of a placement is determined by the cumulative illegality of the placement, overall timing cost, and wirelength cost. The timing cost is used to prevent moves that would increase critical path delay, while the wirelength cost is used to estimate the routability of a placement.

The retiming-aware techniques for general-purpose FPGAs presented so far use heuristics to place retiming registers in the logic blocks of the FPGA. An alternative to placing registers in the logic structure of a general-purpose FPGA is to allocate the registers in the routing structure. In [14], the authors propose a routing algorithm that attempts to move long (and hence critical) routes onto tracks that have registered routing switches. The algorithm exploits the planarity of the target architecture to permute the routes on a registered / unregistered track with those on a compatible unregistered / registered track. An architecturally constrained retiming algorithm is coupled with the routing step to identify tracks that are used by critical routes. All routes on a given critical track are then permuted with a compatible registered track, so that critical routes can go through registered routing switches. After the completion of retiming-aware routing, a final retiming step is performed to achieve a target clock period.

There are two important shortcomings of the retiming-aware routing algorithm presented in [14]. First, the process of permuting routes on to registered tracks may be overly restrictive, since *all* routes on a registered track must go through registered routing switches. While long routes may benefit from an assignment to a registered routing track, other less-critical routes on the track will use up registered switches unnecessarily. Second, the routing algorithm relies on planar FPGA architectures to enable track permutation. Consequently, the algorithm cannot be used to route netlists on non-planar FPGA architectures that have registered routing switches.

In summary, it is clear from sub-section II-A that the area overhead incurred in eliminating the problem of locating pipelining registers is high. At the same time, the heuristic techniques described in II-B are *architecture-specific* solutions that might not be applicable to a range of architecturally diverse pipelined FPGAs. The subject of this paper is the development of an architecture independent pipelining-aware FPGA routing algorithm called PipeRoute. The primary

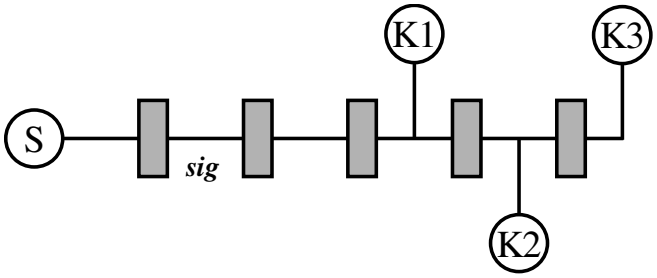

Fig. 1. A multi-terminal pipelined signal.

strength of the PipeRoute algorithm lies in the fact that it is architecture-independent. The algorithm may be used to route netlists on *any* FPGA architecture that can be abstractly represented as a graph consisting of routing and pipelining nodes. Unlike the techniques presented in this section, PipeRoute does not rely on specific architectural features or heuristics to successfully route signals. PipeRoute's architecture adaptability is a direct result of using Pathfinder [6] as the core routing algorithm.

III. Problem Background

The traditional FPGA routing problem is to determine an assignment of signals to limited routing resources while trying to achieve the best possible delay characteristics. Pathfinder [6] is one of the most widely used FPGA routing algorithms. It is an iterative algorithm, and consists of two parts. The *signal* router routes individual signals based on Prim's algorithm, which is used to build a Minimum Spanning Tree (MST) on an undirected graph. The *global* router adjusts the cost of each routing resource at the end of an iteration based on the demands placed on that routing resource during that iteration. During the first routing iteration, signals are free to share as many routing resources as they like. However, the cost of using a shared routing resource is gradually increased during later iterations, and this increase in cost is proportional to the number of signals that share that resource. Thus, this scheme forces signals to negotiate for routing resources. A signal can use a high cost resource if all remaining resource options are in even higher demand. On the other hand, a signal that can take an alternative, lower cost route is forced to do so because of competition for shared resources. Circuits routed using Pathfinder's congestion resolution scheme converge quickly, and exhibit good delay characteristics.

In the case of retimed netlists, the routing problem is different from the conventional FPGA routing problem. This is because a significant fraction of the signals in a netlist are deeply pipelined, and merely building an MST for a pipelined signal is not enough. For example, consider the pipelined signal sig in Fig. 1 that has a source S and sinks K1, K2 and K3. The signal is pipelined in such a way that sink K1 must be delayed 3 clock cycles relative to S, sink K2 must be 4 clock cycles away, and sink K3 must be 5 clock cycles away. A route for sig is valid only if it contains enough pipelining resources to satisfy the clock cycle constraints at every sink. Due to the fact that there are a fixed number of sites in the interconnect where a signal can go through a register, it can be easily seen that a route found for sig by a conventional, pipelining-unaware FPGA router may not go through sufficient registers to satisfy the clock cycle constraint at every sink. Thus, the routing problem for pipelined signals is different from that for unpipelined signals. For a two-terminal pipelined signal, the routing problem is stated as:

Two-terminal N_D Problem: *Let G=(V,E) be an undirected graph, with the cost of each node v in the graph being w_v >= 1. The graph consists of two types of nodes: D-nodes and R-nodes. Let S, K∈V be two R-nodes. Find a path $P_G(S,K)$ that connects nodes S and K, and contains at least N (N ≥ 1) distinct D-nodes, such that $w(P_G(S,K))$ is minimum, where*

$$w(P_G(S,K)) = \sum_{v \in P_G(S,K)} w_v$$

Further, impose the restriction that the path cannot use the same edge to both enter and exit any D-node.

We call a route that contains at least 'N' distinct D-nodes an 'N_D' route. R-nodes represent interconnect wire-segments and the IO pins of logic units in a pipelined FPGA architecture, while D-nodes represent registered switch-points. A registered switch-point (from this point on, we will use the terms 'registered switch-points' and 'registers' interchangeably) can be used to pick up 1 clock cycle delay, or no delay at all. Every node is assigned a cost, and an edge between two nodes represents a physical connection between them in the architecture. The cost of a node is a function of congestion, and is identical to the cost function developed for Pathfinder's NC algorithm [6]. Under this framework, the routing problem for a simpler two-terminal signal is to find the lowest cost route between source and sink that goes through at least N (N ≥ 1) distinct D-nodes (N is the number of clock cycles that separates the source from the sink). Note that in this version a lowest cost route can be self-intersecting i.e. R-nodes can be shared in the lowest cost route. In Appendix A of this paper, we show that the two terminal N_D problem is NP-Complete via a reduction from the Traveling Salesman Problem with Triangle Inequality.

IV. One-Delay (1_D) Router

In the previous section, we pointed out that the problem of finding the lowest cost route between a source and sink that goes through at least N distinct D-nodes is NP-Complete. We now show that a lowest cost route between a source and sink that goes through at least *one* D-node can be found in polynomial time. On a weighted undirected graph, Dijkstra's algorithm is widely used to find the lowest cost route between a source and sink node. The remainder of this section evaluates several modifications of Dijkstra's algorithm that can be used to find a lowest cost 1_D route. Our first modification is *Redundant-Phased-Dijkstra*. In this algorithm, a phase 0 wavefront is launched at the source. When the phase 0 exploration hits a D-node, it is locally terminated there (i.e.

the phase 0 exploration is not allowed to continue through the D-node, although the phase 0 exploration can continue through other R-nodes and runs simultaneously with the phase 1 search), and an independent phase 1 wavefront is begun instead. When commencing a phase 1 wavefront at a D-node, we impose a restriction that disallows the phase 1 wavefront from exiting the D-node along the same edge that was used to explore it at phase 0. This is based on the assumption that it is architecturally infeasible for the D-node that originates the phase 1 wavefront to explore the very node that is used to discover it at phase 0. When a phase 1 wavefront explores a D-node, the D-node is treated like an R-node, and the phase 1 wavefront propagates through the D-node.

If the number of D-nodes that can be explored at phase 0 from the source is 'F', up to F independent phase 1 wavefronts can co-exist during *Redundant-Phased-Dijkstra*. The search space of the phase 1 wavefronts can overlap considerably due to the fact that each R-node in the graph can be potentially explored by up to F independent phase 1 wavefronts. Consequently, the worst-case run-time of *Redundant-Phased-Dijkstra* is F+1 times that of the conventional Dijkstra's algorithm. Since F could potentially equal the total number of interconnect registers in a pipelined FPGA, the worst-case run-time of *Redundant-Phased-Dijkstra* may get prohibitive.

An alternative to *Redundant-Phased-Dijkstra* that can be used to find a lowest cost 1_D route is *Combined-Phased-Dijkstra*. This algorithm attempts to reduce run-time by combining the search space of the phase 1 wavefronts that originate at D-nodes. The only difference between *Redundant-Phased-Dijkstra* and *Combined-Phased-Dijkstra* is that the latter algorithm allows each R-node to be visited only *once* by a phase 1 wavefront. As a consequence, the run-time of *Combined-Phased-Dijkstra* is only double that of Dijkstra's algorithm. In both *Redundant-Phased-Dijkstra* and *Combined-Phased-Dijkstra*, the phase 1 search begins at a cost equal to the path up to the D-node that starts the wavefront. The final route is found in two steps. In the first step, the phase 1 segment of the route is found by backtracing the phase 1 wavefront to the D-node that initiated the wavefront. The phase 0 segment of the route is then found by backtracing the phase 0 wavefront from the D-node back to the source.

A step-by-step illustration of how *Combined-Phased-Dijkstra* works is shown in Figs. 2(a) through 2(e). For the sake of simplicity, assume all nodes in the example graph have unit cost. The source S is explored at phase 0 at the start of the phased exploration. The number 0 next to S in Fig. 2(a) indicates that S has been explored by a phase 0 wavefront. In Fig. 2(b), the neighbors of S are explored by the phase 0 wavefront initiated at S. The 2nd-level neighbors of S are explored by phase 0 in Fig. 2(c), one of which is D-node D1. Note that we make a special note of D1's phase 0 predecessor here, so that we do not explore this predecessor by means of the phase 1 wavefront that is commenced at D1. In Fig. 2(d), the neighbors of D1 (excluding R1) are explored at phase 1. The phase 0 exploration also continues simultaneously, and

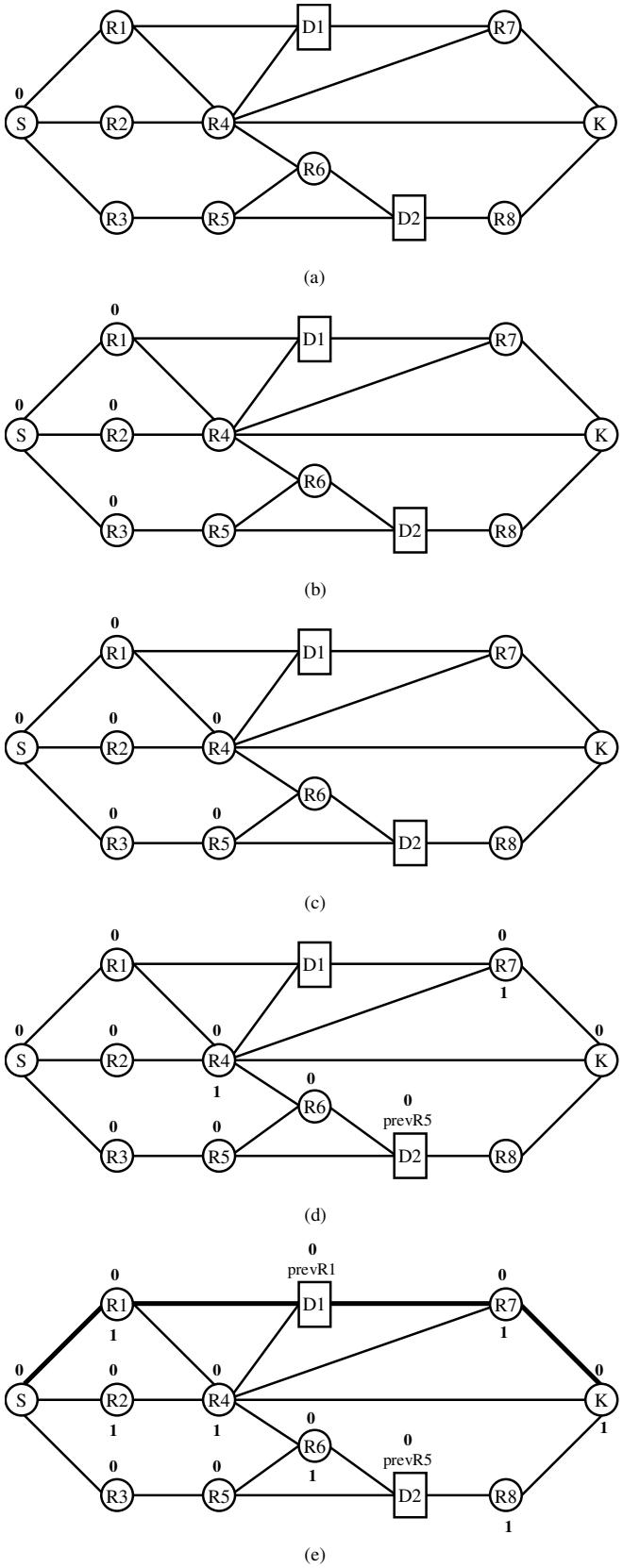

Fig. 2. (a) Phase 0 exploration commences at node S. (b) The neighbors of S are explored at phase 0. (c) 2nd – level neighbors of S are explored at phase 0, and in the process D-node D1 is discovered. (d) D1 starts a phase 1 exploration. The phase 0 exploration continues simultaneously, and D2 is discovered. (e) K is explored by phase 1 wavefront commenced at D1.

note how both phase 0 and phase 1 wavefronts have explored nodes R4 and R7. Finally, in Fig. 2(e), the sink K is explored by the phase 1 wavefront initiated at D1. The route found by *Combined-Phased-Dijkstra* is shown in boldface in Fig. 2(e), and is in fact an optimal route between S and K.

Unfortunately, *Combined-Phased-Dijkstra* fails to find a lowest cost route on some graph topologies. An example of a failure case is shown in Fig. 3. Here the node S is both the source and sink of a signal, and each node is unit cost. *Combined-Phased-Dijkstra* will fail to return to S at phase 1 because R-nodes on each possible route back to S have already been explored by the phase 1 wavefront. In effect, *Combined-Phased-Dijkstra* isolates nodes S, R1, R2, D1 and D2 from the rest of the graph, thus precluding the discovery of any route back to S at all.

The reason for the failure of *Combined-Phased-Dijkstra* is that a node on the phase 1 segment of the lowest cost route is instead explored by a phase 1 wavefront commenced at another D-node. For example, in Fig. 3 we consider the route S-R1-D1-R3-R5-R4-D2-R2-S to be lowest cost. Node R4 is explored by the phase 1 wavefront commenced at D2, thus precluding node R4 from being explored by the phase 1 wavefront started at D1. However, if we slightly relax *Combined-Phased-Dijkstra* to allow each node in the graph to be explored by at most *two* phase 1 wavefronts that are independently started at different D-nodes, then the phase 1 wavefronts started at D1 and D2 will now be able to overlap, thus allowing the lowest cost route to be found.

An important consequence of the nature of the transition from phase 0 to phase 1 at a D-node is shown in Fig. 4. In this case, S is the source of the signal, and K is the sink. Observe that a phase 0 exploration explores D1 from R1. Consequently, the phase 0 exploration is precluded from exploring D1 from R4. This prevents the optimal 1_D route to K from being found.

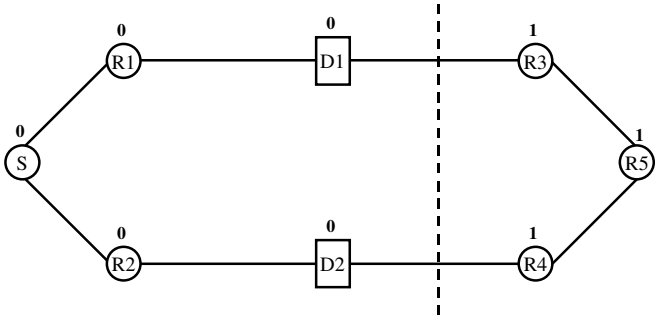

Fig. 3. A case for which phased exploration fails. Observe how the phase 1 exploration has got isolated from the phase 0 exploration.

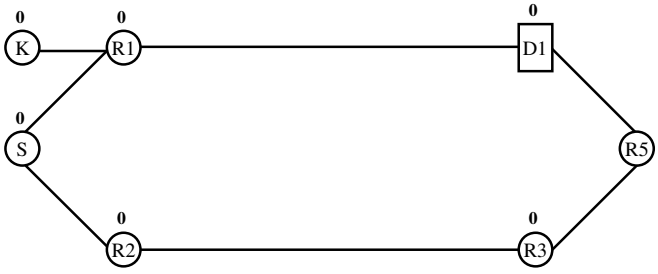

Fig. 4. D1 is explored at phase 0 from R1, thus precluding the discovery of the 1_D path to the sink K.

To address this problem, we allow any D-node to be explored at most two times at phase 0. In Fig. 4, D1 can be explored at phase 0 from R1 and R4, thus allowing the optimal 1_D path S-R2-R3-R4-D1-R1-K to be found.

Fig. 5 shows pseudo-code for the algorithm **`2Combined-Phased-Dijkstra`** that finds an optimal 1_D route between a source **`S`** and sink **`K`**. At the start of the algorithm, a phase 0 exploration is commenced at the source by initializing the priority queue **`PQ`** to **`S`** at phase 0. The phase 0 wavefront is expanded in a manner similar to that of Dijkstra's algorithm. Each time a node **`lnode`** is removed from **`PQ`**, its phase is recorded in the variable **`phase`**. The cost of the path from **`S`** to **`lnode`** is stored in **`path_cost`**. The variable **`node_type`** indicates whether **`lnode`** is an R-node or D-node. The fields **`lnode.num_ex0`** and **`lnode.num_ex1`** record the number of times **`lnode`** has been explored at phase 0 and 1 respectively, and are both initialized to 0. A node is marked *`finally_explored`* at a given phase when it is no longer possible to expand a wavefront through that node at the given phase. For each **`lnode`** that is removed from **`PQ`**, the following possibilities exist:

- **`phase`**`==0` and **`node_type`** is R-node: R-nodes can be explored at phase 0 only once, and thus **`lnode`** is *`finally_explored`* if **`x0`**`==1`. **`AddNeighbors`**`(`**`PQ,lnode,path_cost,p`**`)` is used to add the neighbors of **`lnode`** to **`PQ`** at phase **`p`**, where **`p`**`==0` in this case.
- **`phase`**`==0` and **`node_type`** is D-node: D-nodes can be explored at phase 0 twice, and thus **`lnode`** is marked *`finally_explored`* if **`x0`**`==2`. A phase 1 exploration is begun at this D-node by adding its neighbors to **`PQ`** at phase 1.
- **`phase`**`==1`: Since both R-nodes and D-nodes can be explored twice at phase 1, **`lnode`** is marked *`finally_explored`* at phase 1 if **`x1`**`==2`. If we are not done (i.e. **`lnode`** is not the sink **`K`**) the neighbors of **`lnode`** are added to **`PQ`** at phase 1.

A. Proof Of Optimality

The optimality of `2Combined-Phased-Dijkstra` can be demonstrated by means of a proof by contradiction in which we show that `2Combined-Phased-Dijkstra` will always find an optimal 1_D path between S and K, if one exists. Before presenting a sketch of the proof, we introduce some terminology. `2Combined-Phased-Dijkstra` explores multiple paths through the graph via a modification to Dijkstra's algorithm. We state that the algorithm explores a path "P" up to a node "N" if the modified Dijkstra's search, in either phase 0 or phase 1, reaches node "N" and the search route to this node is identical to the portion of the path P from the source to node N. Further, a path A is "more explored" than path B if the cost of the path on A from the source to A's last explored point is greater than the cost of the path on B

```
2Combined-Phased-Dijkstra(S,K){
  Init PQ to S at phase 0;
  LOOP{
    Remove lowest cost node lnode from PQ;
    if(lnode == NULL){
      1D path between S and K does not exist;
      return 0;
    }
    if(lnode is finally_explored at phase 0 and phase 1)
      continue;
    path_cost = cost of path from S to lnode;
    phase = phase of lnode;
    node_type = type of lnode;
    if(phase == 0){
      lnode.num_ex0++;
      x0 = lnode.num_ex0;
    }
    else{
      lnode.num_ex1++;
      x1 = lnode.num_ex1;
    }
    if(phase  == 0){
      if(node_type == R-node){
        if(x0 == 1)
          Mark lnode finally_explored at phase 0;
        AddNeighbors(PQ,lnode,path_cost,0);
      }
      else{
        if(x0 == 2)
          Mark lnode finally_explored at phase 0;
        AddNeighbors(PQ,lnode,path_cost,1);
      }
    }
    else{
      if(lnode == K)
        return backtraced 1D path from S to K;
      else{
        if(x1 == 2)
          Mark lnode finally_explored at phase 1;
        AddNeighbors(PQ,lnode,path_cost,1);
      }
    }
  }END LOOP
}

AddNeighbors(PQ,lnode,path_cost,p){
  Foreach neighbor neb_node of lnode{
    neb_cost = cost of neb_node;
    neb_path_cost = neb_cost + path_cost;
    Add neb_node to PQ with phase p at cost neb_path_cost;
  }
}
```

Fig. 5. Pseudocode for 2Combined-Phased-Dijkstra.

from the source to B's last explored point. For purposes of the proof sketch, we define the "goodness" of a path in the following way:

1. If the cost of one path is lower than another's, it is "better" than the other. Thus, an optimal path is always better than a non-optimal path.
2. If the costs of two paths C and D are the same, then C is "better" than D if C is more explored than D.

From these definitions, the "best" path is an optimal path. If there is more than one optimal path, the best path is the most explored optimal path.

***Initial Assumption*:** Assume that Fig. 6 shows the *most explored* optimal 1_D path between S and K. In other words, the path shown in the figure is the best 1_D path between S and K, with a single clock-cycle delay picked up at D-node D_L. Note that there are no D-nodes on the path S-D_L, although there could be multiple D-nodes on D_L-K. This is because we assume that in case the best 1_D path between S and K goes through multiple D-nodes, then the D-node nearest S is used to pick up one clock-cycle delay.

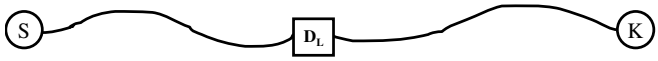

Fig. 6. The initial assumption is that the most explored lowest cost 1_D route between S and K goes through D-node D_L.

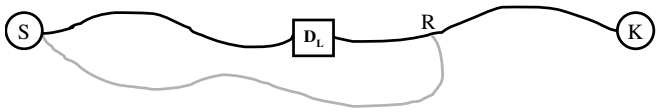

Fig. 7. Representation of a path from S to node R shown in gray.

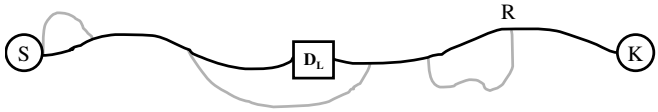

Fig. 8. The path from S to R could actually intersect with the paths S-D_L and D_L-K.

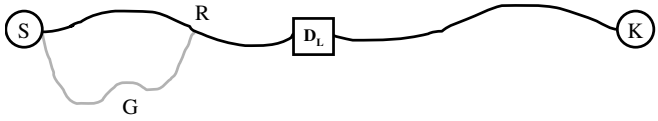

Fig. 9. The case in which an R-node on the path S-D_L gets explored at phase 0 along some other path.

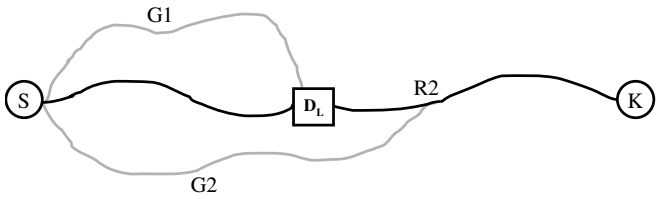

Fig. 10. D_L gets explored at phase 0 along paths S-G1-D_L and S-G2-R2-D_L.

Although it appears that the paths S-D_L and D_L-K in Fig. 6 are non-intersecting, note that the R-nodes on the path S-D_L can in fact be reused in the path D_L-K. In all diagrams in this section, we use the convention of showing paths without overlaps (Fig. 7), even though they may actually overlap (Fig. 8). Our proof does not rely on the extent of intersection between hypothetical paths (which are always shown in gray) and the known best 1_D path.

There are three distinct cases in which `2Combined-Phased-Dijkstra` could fail to find the best path S-D_L-K shown in Fig. 6:

- **CASE 1:** An R-node on the path S-D_L gets explored at phase 0 along a path other than S-D_L.
- **CASE 2:** The D-node D_L gets explored at phase 0 along two paths other than S-D_L.
- **CASE 3:** A node on the path D_L-K gets explored at phase 1 along two paths other than D_L-K.

Fig. 9 shows why **CASE 1** can never occur. For **CASE 1** to occur, the cost of the gray path S-G-R would have to be less than or equal to the cost of path S-R. In this case, the path S-G-R-D_L-K would be better than the known best path, which is a contradiction of our initial assumption.

Fig. 10 shows an instance of **CASE 2**. The cost of each of the paths S-G1-D_L and S-G2-R2-D_L is less than or equal to the cost of path S-D_L. In this case, the path S-G1-D_L-R2-K would

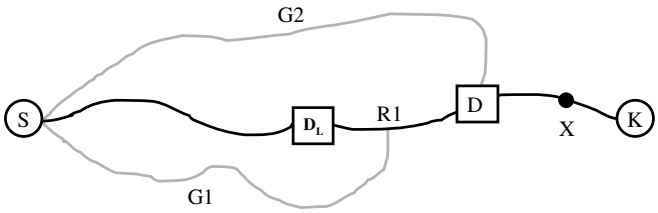

Fig. 11. Node X can get explored at phase 1 along either S-G2-D-X or S-G1-R1-D-X.

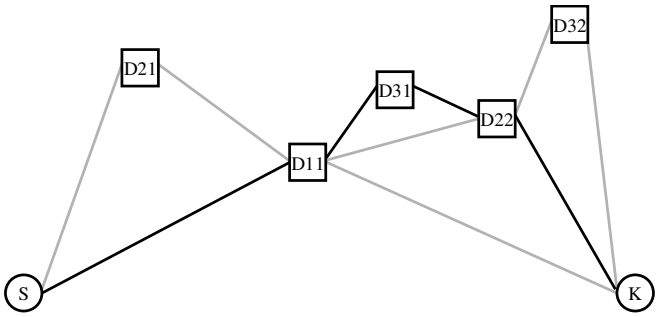

Fig. 12. Building a 3_D route from 1_D routes.

be better than the known best path S-D_L-K, thus contradicting our initial assumption.

Fig. 11 illustrates an example of **CASE 3**, in which a node X on the path D_L-K gets explored at phase 1 along two paths other than D_L-K. There are two possibilities here:

- The cost of path S-G1-R1-D-X is less than or equal to the cost of the path to X along the known best path. In this case, the path S-G1-R1-D-X-K would be better than the known best path, which is a contradiction of our initial assumption.
- The cost of path S-G2-D-X is less than or equal to the cost of the path to X along the known best path. This means that the path S-G2-D-X-K is better than the known best path, which contradicts our initial assumption.

A more detailed case-by-case analysis of the proof of optimality of 2Combined-Phased-Dijkstra can be found in [11]. In this study, we enumerate all the possible sub-cases of CASE 1, CASE 2 and CASE 3 and separately show that each of the sub-cases contradicts our initial assumption. Consequently, none of CASE 1, CASE 2 or CASE 3 can occur, implying that 2Combined-Phased-Dijkstra is optimal.

V. N-Delay (N_D) Router

In this section, we present a heuristic that uses the optimal 1_D router to build a route for a two terminal N_D signal. This heuristic greedily accumulates D-nodes on the route by using 1_D routes as building blocks. In general, an N_D route is recursively built from an $(N-1)_D$ route by successively replacing each segment of the $(N-1)_D$ route by a 1_D route and then selecting the lowest cost N_D route. Fig. 12 is an abstract illustration of how a 3_D route between S and K is found. In the first step, we find a 1_D route between S and K, with D11 being the D-node where we pick up a register. At this point, we increment the sharing cost [6] of all nodes that constitute the route S-D11-K. In the second step, we find two 1_D routes, between S and D11, and D11 and K. The sequence of sub-steps in this operation is as follows:

- Decrement sharing cost of segment S-D11.
- Find 1_D route between S and D11 (S-D21-D11). Store the cost of route S-D21-D11-K.
- Restore segment S-D11 by incrementing the sharing cost of segment S-D11.
- Decrement sharing cost of segment D11-K.
- Find 1_D route between D11 and K (D11-D22-K). Store the cost of route S-D11-D22-K.
- Restore segment D11-K by incrementing the sharing cost of segment D11-K.
- Select the lowest cost route, either S-D21-D11-K or S-D11-D22-K.

Suppose the lowest cost 2_D route is S-D11-D22-K. We rip up and decrement sharing due to the segment D11-K in the original route S-D11-K, and replace it with segment D11-D22-K. Finally, we increment sharing of the segment D11-D22-K. The partial route now is S-D11-D22-K. The sequence of sub-steps in step three is similar. Segments S-D11, D11-D22 and D22-K are successively ripped up, replaced with individual 1_D segments, and for each case the cost of the entire 3_D route between S and K is stored. The lowest cost route is then selected. In Fig. 12, the 3_D route that is found is shown in dark lines, and is S-D11-D31-D22-K.

The number of 1_D explorations launched for the 3_D route that we just discussed is 1 + 2 + 3 = 6. For the general N_D case, the number of 1_D explorations launched is 1 + 2 + ... + N = N(N+1)/2.

VI. Multi-Terminal Router

The previous section described a heuristic that uses optimal 1_D routes to build a two-terminal N_D route. The most general type of pipelined signal is a multi-terminal pipelined signal. A multi-terminal pipelined signal has more than one sink, and the number of registers separating the source from each sink could differ across the set of sinks. A simple example of a multi-terminal pipelined signal ***sig*** was shown in Fig. 1. The sinks K1, K2 and K3 must be separated from the source S by 3, 4 and 5 registers respectively. We will now demonstrate how a route for a multi-terminal signal can be found by taking advantage of the 1_D and N_D routers that were discussed in Sections IV and V.

In a manner similar to the Pathfinder algorithm, the routing tree for a multi-terminal pipelined signal is built one sink at a time. Each sink is considered in non-decreasing order of register separation from the source of the signal. The multi-terminal router starts by finding a route to a sink that is the least number of registers away from the source. Since finding a route to the first sink is a two-terminal case, we use the two-terminal N_D router to establish a route between the source and first sink. The remainder of this section examines the task of expanding the route between the source and the first sink to include all other sinks.

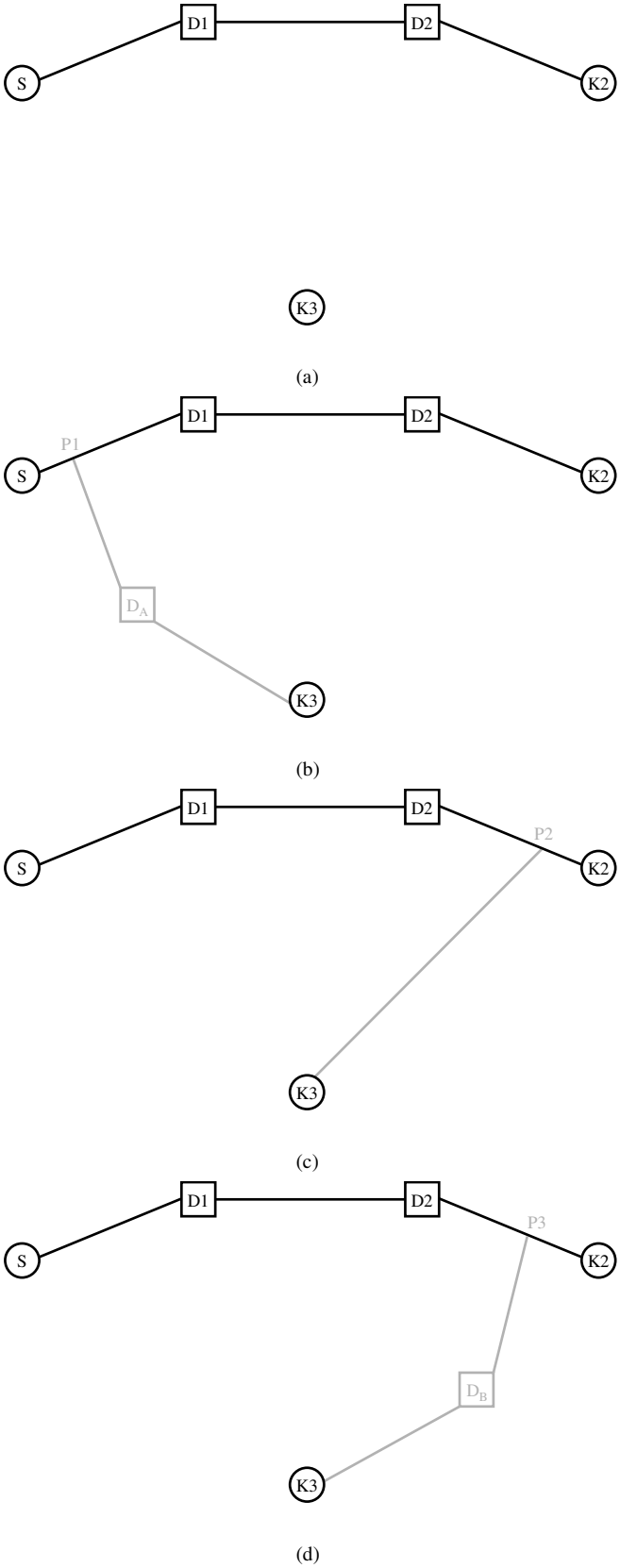

Fig. 13. (a) 2_D route to K2 using the two-terminal N_D router. S-D1-D2-K2 is the *partial_routing_tree*. (b) 1_D route to K3. P1-D_A-K3 is found by launching a 1_D exploration that starts with segment S-D1 at phase 0 and segment D1-D2 at phase 1. P1-D_A-K3 is the *surviving_candidate_tree*. (c) 2_D route to K3. P2-K3 is now the *surviving_candidate_tree*. (d) P3-D_B-K3 is the final *surviving_candidate_tree*, and this tree is joined to the *partial_routing_tree* S-D1-D2-K2 to complete the route to K3.

We explain the multi-terminal router via a simple example. Assume a hypothetical signal that has a source S and sinks K2 and K3. K2 must be separated from S by 2 registers, whereas K3 must be separated by 3 registers. Sink K2 is considered first, and the N_D router is used to find a 2_D route between S and K2. In Fig. 13(a), the route S-D1-D2-K2 represents the 2_D route between S and K2, and constitutes the *partial_routing_tree* of the signal. In general, the *partial_routing_tree* of a multi-terminal pipelined signal can be defined as the tree that connects the source to all sinks that have already been routed.

After a route to K2 is found, the router considers sink K3. As was the case in the N_D router, we accumulate registers on the route to K3 one register at a time. Thus, we start by finding a 1_D route to K3, then a 2_D route, and finally a 3_D route to K3. It can be seen that a 1_D route to K3 can be found either from the 0_D segment S-D1 by going through another D-node, or from the 1_D segment D1-D2 directly. However, it is not necessary to launch independent wavefronts from segments S-D1 and D1-D2. This is because both wavefronts can be combined into a single 1_D search in which segment S-D1 constitutes the starting component of the phase 0 wavefront, and segment D1-D2 constitutes the starting component of the phase 1 wavefront. Setting up the 1_D search in such a way could find a 1_D path from S-D1 or a 0-delay path from D1-D2, depending on which is of lower cost. Assume that P1-D_A-K3 is the 1_D route found to K3 (Fig. 13(b)). After the 1_D route to K3 is found, the sharing cost of the nodes that constitute P1-D_A-K3 is incremented. The segment P1-D_A-K3 is called the *surviving_candidate_tree*. The *surviving_candidate_tree* can be defined as the tree that connects the sink (K3 in this case) under consideration to some node in the *partial_routing_tree* every time an N_D route ($1 \le N \le 3$ in this case) to the sink is found. Thus, a distinct *surviving_candidate_tree* results immediately after finding the 1_D, 2_D, and 3_D routes to K3.

Next, we attempt to find a 2_D route to K3. Before explaining specifics, it is important to point out here that while finding an N_D route to a sink we try two options. The first is to alter the *surviving_candidate_tree* to include an additional D-node as was done in the two terminal N_D router. The second option is to use the N_D and $(N-1)_D$ segments in the *partial_routing_tree* together to start a 1_D exploration. The lower cost option is chosen, and this becomes the new *surviving_candidate_tree*.

For finding a 2_D route to K3, we first modify P1-D_A-K3 to include another D-node much in the same way that a two terminal 2_D route is built from an already established 1_D route (Section V). The segments P1-D_A and D_A-K3 are each separately replaced by optimal 1_D routes, and the lowest cost route is stored. To evaluate the second option, we rip up the segment P1-D_A-K3 (Fig. 13(b)) and launch a 1_D search using segments D1-D2 at phase 0 and D2-K2 at phase 1. The cost of the resultant 1_D route is also stored. The lower cost route amongst the two options is chosen, and the sharing cost of the nodes that constitute this route is incremented. This selected route becomes the new *surviving_candidate_tree*. In Fig.

```
Multi-Terminal-Router (Net){
  P_RT = ϕ; C_RT = ϕ;
  Sort elements of S_K in non-decreasing order
    of Dnode-separation from Src_Net;
  Use the two-terminal N_D-Router to find
    route R from Src_Net to S_K[1];
  Add R to the partial routing tree P_RT;
  Foreach i in 2…|S_K| {
    k_i = S_K[i];
    d_i = num Dnodes between Src_Net and k_i;
    Foreach j in 1…d_i {
      Use the two-terminal N_D-Router to find
        a j_D route called RN_j by altering the
        (j-1)_D route contained in C_RT;
      Use 2Combined-Phased-Dijkstra to build
        a j_D route called RD_j from the (j-1)_D
        and j_D segments of the route contained
        in P_RT;
      if cost(RD_j) < cost(RN_j) {
        C_RT = RD_j;
      }
      else {
        C_RT = RN_j;
      }
    }
    Add surviving candidate tree C_RT to partial
      routing tree P_RT;
  }
  return the route contained in P_RT;
}
```

Fig. 14. Pseudocode for the multi-terminal routing algorithm.

13(c), assume that the lower cost route that is selected is the segment P2-K3 shown in gray.

Finally, the segment P2-K3 is ripped up and a 1_D exploration from the segment D2-K2 is launched at phase 0 to complete the 3_D route to K3 (Fig. 13(d)).

Fig. 14 presents pseudo-code for the multi-terminal routing algorithm. **Net** is the multi-terminal signal that is to be routed. Without loss of generality, we assume that **Net** has at least two sinks, and each sink is separated from **Net**'s source by at least one D-node. $\mathbf{P_{RT}}$ contains the *partial_routing_tree* during the execution of the algorithm, and $\mathbf{C_{RT}}$ contains the *surviving_candidate_tree*. $\mathbf{Src_{Net}}$ is the source of the signal **Net**, while $\mathbf{S_K}$ is an array that contains **Net**'s sinks. $\mathbf{N_D}$**-Router** is the N-Delay router presented in Section V.

VII. MULITPLE REGISTER SITES

The PipeRoute algorithm described in Sections IV, V and VI assumes that register sites (D-nodes) in the interconnect structure can only provide zero or one register. Also, the algorithm does not address the fact that the IO terminals of logic units may themselves be registered. Since a number of pipelined FPGA architectures [3] – [4], [16] do in fact provide registered IO terminals and multiple-register sites in the interconnect structure, we developed a greedy pre-processing heuristic that attempts to maximize the number of registers that can be acquired at registered IO terminals and multiple-register sites. We present the details of this heuristic in three parts:

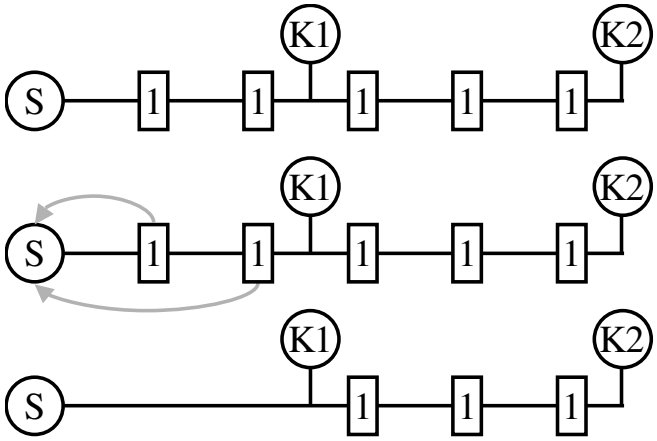

Fig. 15. Assuming that S can provide up to three registers locally, both the registers between S and K1 can be picked up at S.

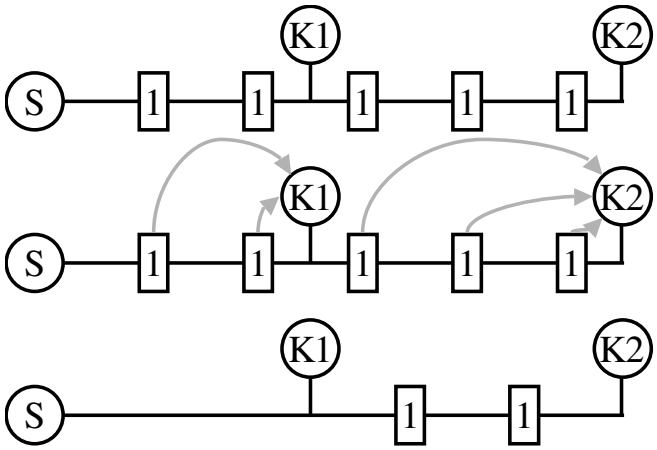

Fig. 16. Assuming that the sinks K1 and K2 can locally provide up to three registers, both registers between S and K1 and three of the five registers between S and K2 can be picked up locally at the respective sinks.

A. *Logic Units with Registered Outputs*

We try to greedily pick up the maximum allowable number of registers at the source of each pipelined signal. The maximum number of registers that can be picked up at the source is capped by the sink that is separated by the least number of registers from the source. Consider the example in Fig. 15. The pipelined signal shown has a source S and two sinks K1 and K2 that must be separated from S by two and five registers respectively. Assuming that up to three registers can be turned on at S, both registers that separate S and K1 can be picked up at S itself, thus eliminating the need to find a 2_D route between S and K1 in the interconnect structure. Instead, we now only need to find a simple lowest-cost route from S to K1, and a 3_D route to K2.

B. *Logic units with Registered Inputs*

In this case, we push as many registers as possible into each sink of a pipelined signal. In Fig. 16, if we again assume that each sink can provide up to three registers locally, both registers between S and K1 can be moved into K1, while three registers between S and K2 can be moved into K2. This leaves us with the task of finding a simple lowest-cost route to K1

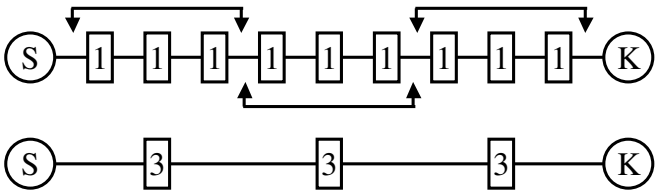

Fig. 17. Finding a 9_D route between S and K can effectively be transformed into a 3_D pipelined routing problem.

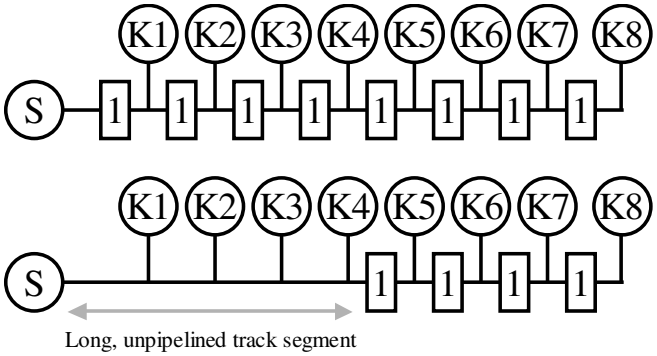

Fig. 18. Pushing registers from the interconnect structure into functional unit inputs sometimes results in long, unpipelined track segments.

and a 2_D route to K2.

C. Multiple-Register Sites in the Interconnect Structure

Multiple-register sites in the interconnect structure provide an opportunity to significantly improve the routability of pipelined signals. In Fig. 17 for example, if we assume that each register site (D-node) in the interconnect can provide up to three registers, the task of finding a two terminal 9_D route simplifies to finding a route that with at least three D-nodes. For a multi-terminal pipelined signal, every time an N_D route to the new sink is to be found, we use all existing N_D, $(N-1)_D$, $(N-2)_D$, and $(N-3)_D$ segments in the current partially built routing tree to start an exploration that finds a single D-node. Since each D-node can be used to pick up between zero and three registers, we use all segments within the current, partially built routing tree that are less than or equal to three registers away from the new sink.

The intuition behind the development of the greedy heuristic in sub-sections VII-A, VII-B and VII-C is to aggressively *reduce* the number of register-sites that need to be found in the interconnect structure. The heuristic is clearly routability-driven, since reductions in the number of interconnect registers favorably impact the routability of pipelined signals. Due to the finite nature of an FPGA's interconnect structure, any place-and-route heuristic must consider routability to ensure that a placement can be successfully routed.

A shortcoming of the greedy heuristic is that long segments of a pipelined signal may get unpipelined because of the removal of registers from the interconnect structure. This phenomenon is illustrated in Fig. 18. Assume that a maximum of four registers can be picked up at the sinks K1 – K8. In this case, one interconnect register will be moved into K1, two into K2, three into K3, and four into K4-K8. This process effectively unpipelines a long segment, which in turn may

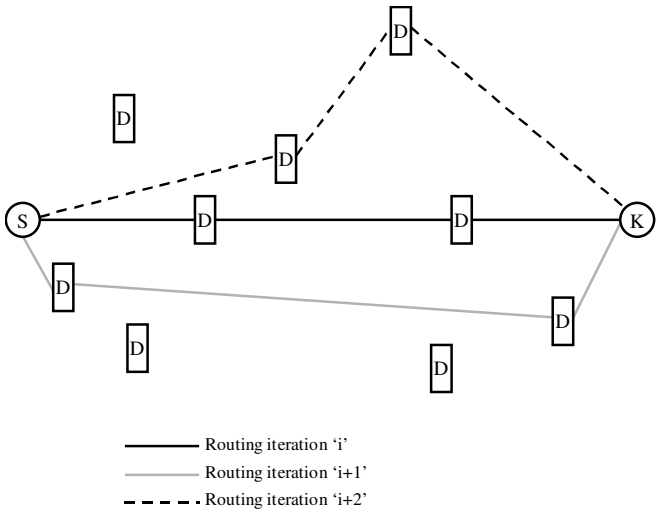

Fig. 19. The route between source S and sink K of a signal may go through different D-nodes at the end of successive routing iterations. Also, since each D-node on the route is used to pick up a register, different segments on the route may be at different criticalities.

increase the critical path delay of a netlist.

To balance register-to-register interconnect delay, it might be necessary to perform a post-routing register redistribution step. Specifically, if the number of interconnect register-sites in the route for a two-terminal pipelined signal is *greater* than the minimum required, then the pipelining registers may be reallocated along the route to balance path delays.

VIII. Timing-Aware Pipelined Routing

Since the primary objective of pipelined FPGAs is the reduction of clock cycle time, it is imperative that a pipelined routing algorithm maintains control over the criticality of pipelined signals during routing. In making PipeRoute timing aware, we draw inspiration from the Pathfinder algorithm. While routing a signal, Pathfinder uses the criticality of the signal in determining the relative contributions of the congestion and delay terms to the cost of routing resources. If a signal is near critical, then the delay of a routing resource dominates the total cost of that resource. On the other hand, if the signal's criticality is considerably less than the critical path, the congestion on a routing resource dominates.

In the case of pipelined routing, the signal's route may go through multiple D-nodes. Consequently, the routing delay incurred in traversing the route from source to sink may span multiple clock cycles. Also, the location of D-nodes on the route may be different across routing iterations. This is because PipeRoute may have to select different routes between the source and sink of a signal to resolve congestion. In Fig. 19 for example, the 2_D route between S and K may go through *different* D-nodes at the end of iterations i, i+1 and i+2 respectively.

To address these problems, we treat D-nodes like normal registers during the timing analysis at the end of a routing iteration. Once the timing analysis is complete, we are faced with making a guess about the overall criticality of a pipelined

signal. Note that different segments of a pipelined signal's route could be at different criticalities (Fig. 19). Our solution is to make a pessimistic choice. Since we know the individual criticalities of signals sourced at each D-node, we make the criticality of the pipelined signal equal to the criticality of the most critical segment on the route. Thus, when the pipelined signal is routed during the next iteration, the most critical segment of the signal's previous route determines the delay cost of routing resources.

IX. Target Pipelined Architecture

In this section we describe features of the pipelined FPGA architecture (RaPiD [3] – [4]) that we used in our experiments. The RaPiD architecture is targeted to high-throughput, compute-intensive applications like those found in DSP. Since such applications are generally pipelined, the RaPiD datapath and interconnect structures include an abundance of registers. The 1-Dimensional (1-D) RaPiD datapath (Fig. 20) consists of coarse-grained logic units that include ALUs, multipliers, small SRAM blocks, and general purpose registers (hereafter abbreviated GPRs). Each logic unit is 16 bits wide. To support pipelining in the logic structure, a register bank is provided at each output of a logic unit. The output register bank can be used to acquire between 0 – 3 registers.

The interconnect structure consists of 1-D routing tracks that are also 16 bits wide. There are two types of routing tracks: short tracks and long tracks. Short tracks are used to achieve local connectivity between logic units, whereas long tracks traverse longer distances along the datapath. In Fig. 20, the uppermost five tracks are short tracks, while the remaining tracks are long tracks. A separate routing multiplexer is used to select the track that drives each input of a logic unit. Each output of a logic unit can be configured to drive multiple tracks by means of a routing demultiplexer.

The long tracks in the RaPiD interconnect structure are segmented by means of bus connectors (shown as empty boxes in Fig. 20 and abbreviated BCs). BCs serve two roles in the RaPiD interconnect structure. First, a BC serves as a buffered, bidirectional switch that facilitates the connection between two long-track segments. Second, a BC serves the role of an interconnect register site or D-node. RaPiD provides the option of picking up between 0 – 3 registers at each BC. The total number of BCs determines the number of registers that can be acquired in the interconnect structure.

While BCs are used as registered, bidirectional switches that connect segments on the same long track, GPRs can be used to switch tracks. A GPR's input multiplexer and output demultiplexer allow a connection to be formed between arbitrary tracks. At the end of a placement phase, all unoccupied GPRs are included in the routing graph as unregistered switches. The ability to switch tracks provides an important degree of flexibility while routing netlists on the RaPiD architecture.

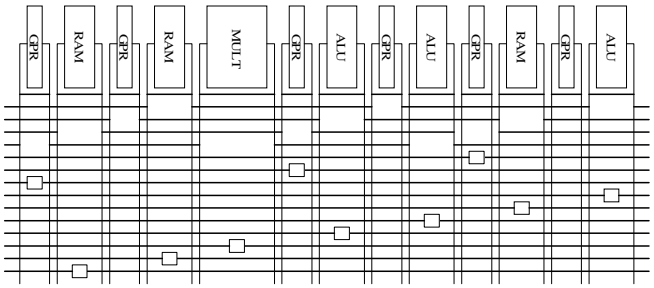

Fig. 20. An example of a RaPiD architecture cell. Several RaPiD cells can be tiled together to create a representative architecture.

X. Placement Algorithm

The placement of a netlist is determined using a Simulated Annealing [5], [7] algorithm. The cost of a placement is formulated as a linear function of the maximum and average *cutsize*, where *cutsize* is the number of signals that need to be routed across a vertical partition of the architecture for a given placement. Since the RaPiD interconnect structure provides a fixed number of routing tracks, the cost function must be sensitive to changes in maximum cutsize. At the same time, changes in average cutsize also influence the cost of a placement. This is because average cutsize is a direct measure of the total wirelength of a placement.

Pipelining information is included in the cost of a placement by mapping each pipelining register (a *pipelining* register is a register that must be mapped to an interconnect register) in the netlist to a unique BC in the interconnect structure. Our high-level objective in mapping pipelining registers to BCs is to place netlist components such that the router is able to find a sufficient number of BCs in the interconnect structure while routing pipelined signals. A more detailed discussion of the placement algorithm can be found in [9], and [11].

Since pipelining registers are explicitly placed, it might be possible to solve the register allocation problem during placement. In general, the pipelining registers in a netlist could be mapped to registered switch-points in the architecture, and a simulated annealing placement algorithm could determine a placement of the pipelining registers. After the placement phase, a conventional FPGA router (Pathfinder) could be used to route the signals in the netlist. While this approach is attractive for its simplicity and ease of implementation, it has a serious shortcoming. A placement of a netlist that explicitly maps pipelining registers to registered switch-points *eliminates* portions of the routing graph. This is because a registered switch-point that is occupied by a particular pipelining register cannot be used by signals other than the signals that connect to that pipelining register. As a consequence, the search space of a conventional FPGA router is severely limited, and this results in solutions of poor quality.

To validate our hypothesis, we ran an experiment on a subset of the benchmark netlists. The objective of the experiment was to find the size of the smallest RaPiD array needed to route (using a pipelining-unaware router Pathfinder) placements produced by the algorithm described in this

TABLE I.
OVERHEAD INCURRED IN USING A PIPELINING-UNAWARE ROUTER (PATHFINDER) TO ROUTE NETLISTS.

NETLIST	NORM. AREA
firtm	1
sobel	1
fft16	1.6
imagerapid	FAIL
cascade	FAIL
matmult4	FAIL
sort_g	FAIL
sort_rb	FAIL
firsymeven	FAIL

section. Note that pipelining registers were explicitly mapped to BCs in the interconnect structure, and the post-placement routing graph was modified to reflect the assignment of pipelining registers to BCs.

Table I presents the results of this experiment. Column 1 lists the netlists in our benchmark set. Column 2 lists the minimum-size array required to route each netlist using Pathfinder. The entries in column 2 are normalized to the minimum-size RaPiD array needed to route the netlists using PipeRoute. A "FAIL" entry in column 2 means that the netlist could not be routed on any array whose normalized size was between 1.0 – 2.0. Table I shows that Pathfinder was unable to route a majority of netlists on arrays that had double the number of logic and routing resources needed to route the placements using PipeRoute. This result clearly showed that pipelining register allocation is best done during the routing phase.

XI. EXPERIMENTAL SETUP AND BENCHMARKS

The set of benchmark netlists used in our experimentation includes implementations of FIR filters, sorting algorithms, matrix multiplication, edge detection, 16-point FFT, IIR filtering and a camera imaging pipeline. While selecting the benchmark set, we included a diverse set of applications that were representative of the domains to which the RaPiD architecture is targeted. We also tried to ensure that the benchmark set was not unduly biased towards netlists with too many or too few pipelined signals. Table II lists statistics of the application netlists in our benchmark set. Column 1 lists the netlists, column 2 lists the total number of nets in each netlist, column 3 lists the percentage of nets that are pipelined, column 4 lists the maximum number of registers needed between any source-sink terminal pair in the netlist (this number is similar to the latency of the application), and column 5 lists the average number of registers needed across all source-sink terminal pairs in the netlist.

While the size of the netlists in Table II might seem small, remember that a single pipelined signal represents *multiple* routing problems. An example of a pipelined signal in the netlist *sort_rb* has 38 sinks. The number of registers that separate the 38 sinks from the source is evenly distributed between 0 registers and 35 registers. Although this signal is counted as a single signal in Table II, finding a route for this signal may require *hundreds* of individual routing searches. Thus, routing the pipelined signals in the benchmark netlists clearly represents a problem of reasonable complexity. Also note that RaPiD is a coarse-grained architecture. Thus, a single net represents a 16-bit bus.

TABLE II.
BENCHMARK APPLICATION NETLIST STATISTICS.

NETLIST	NUM NETS	% PIPELINED	MAX DEPTH	AVG DEPTH
firtm	158	3	16	5.5
fft16	94	29	3	0.74
cascade	113	40	21	3.88
matmult4	164	44	31	4.62
sobel	74	44	5	1.44
imagerapid	101	51	12	3.46
firsymeven	95	54	31	6.98
sort_g	70	65	35	4.98
sort_rb	63	71	35	5.42

Applications are mapped to netlists using the RaPiD compiler [3], and the architecture is represented as an annotated structural Verilog file. Area models for the RaPiD architecture are derived from a combination of the current layout of the RaPiD cell, and transistor-count models. The delay model is extrapolated from SPICE simulations. Each netlist is placed using the algorithm presented in Section X. The placement algorithm places pipelining registers into BC positions in order to model the demands of pipelining. However, the BC assignments are removed before routing to allow PipeRoute full flexibility in assigning pipelining registers.

The netlists are routed using timing-aware PipeRoute that can handle multiple-register IO and interconnect sites. A netlist is declared unrouteable on an architecture of a given size (where size is the number of RaPiD cells that constitute the architecture) if PipeRoute fails to route the netlist in 32 tracks.

XII. RESULTS

The objective of our first experiment (**Experiment 1**) was to quantify the area overhead incurred in routing the benchmark netlists on an *optimized* RaPiD architecture [12]. The logic units in this architecture have registered input terminals (the original RaPiD architecture in Section IX has registered output terminals), and between 0 – 3 registers can be acquired at each input terminal and BC. Also, unlike the original RaPiD architecture, the optimized RaPiD architecture in [12] has nine GPRs in every RaPiD cell.

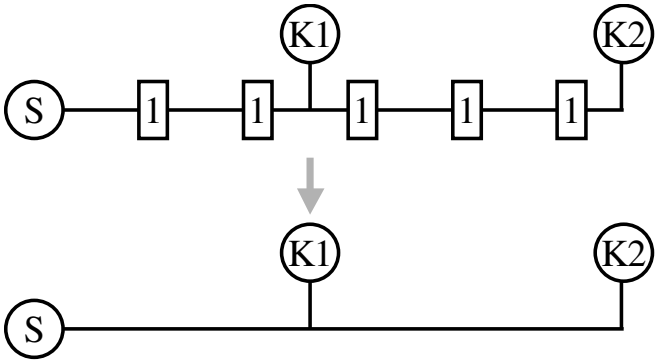

Fig. 21. Unpipelining a pipelined signal.

We acquired area numbers by running the entire set of benchmarks through two place-and-route flows. The first is a pipelining-unaware flow that treats netlists as if they were unpipelined. Specifically, all pipelined signals in a netlist are treated like normal, unpipelined signals (Fig. 21). The pipelining-unaware placement tool attempts to reduce only maximum and average cutsize (Section X). The pipelining-unaware router attempts only connectivity routing, since there are no registers to be found in the interconnect structure. The pipelining-unaware place and route flow provides a *lower-bound* on the size of the smallest architecture needed to successfully route the benchmark netlists. This is because the best area that we can expect from a pipelining-*aware* flow would be no better than a pipelining-unaware flow that ignores pipelining altogether.

The second flow is the pipelining-aware flow described in this paper. Netlists are placed using the algorithm described in Section X, and routed using purely congestion-driven PipeRoute. For both approaches, we recorded the area of the smallest architecture required to successfully route each netlist. Table III lists the smallest areas found for each benchmark netlist using both pipelining-aware and pipelining-unaware flows. The area overhead varied between 0% (for the netlists *fft16*, *matmult4* and *sobel*) and 44% for the netlist *firsymeven*. Overall, the geometric mean of the overhead incurred across the entire benchmark set was 18%. We regard this a satisfactory result, since a pipelining-aware flow incurs less than a 20% penalty over a likely unachievable lower-bound.

The objective of our second experiment (**Experiment 2**) was to investigate the performance of timing-aware PipeRoute vs. timing-unaware PipeRoute. For both approaches, we separately obtained the post-route critical path delays of benchmark netlists routed on the smallest possible RaPiD architecture. Table IV shows the results that we obtained. Across the entire benchmark set, timing-aware PipeRoute produced an 8% improvement in critical path delay compared to timing-unaware PipeRoute.

Our final experiment (**Experiment 3**) was to study whether there is any relationship between the fraction of pipelined signals in a benchmark netlist and the area overhead incurred in successfully routing the netlist on a minimum size architecture. The area overhead is a measure of the pipelining 'difficulty' of a netlist and is quantified in terms of the

TABLE III.
EXPERIMENT 1 – AREA COMPARISON BETWEEN PIPELINING-AWARE AND PIPELINING-UNAWARE PLACE AND ROUTE FLOWS.

NETLIST	PIPELINING UNAWARE AREA (um^2)	PIPELINING AWARE AREA (um^2)
sort_g	3808215	5183743
sort_rb	3808215	5752143
fft16	5712322	5712322
imagerapid	6664376	8025125
firsymeven	6897972	9949143
firtm	7257201	7616430
matmult4	7616430	7616430
cascade	7616430	8753230
sobel	9039119	9039119
GEOMEAN	**6247146**	**7347621**

TABLE IV.
EXPERIMENT 2 – DELAY COMPARISON BETWEEN TIMING-AWARE AND TIMING-UNAWARE PIPEROUTE.

NETLIST	TIMING-UNAWARE (ns)	TIMING-AWARE (ns)
firtm	6.73	6.63
matmult4	8.27	8.57
sort_rb	9.65	12.61
firsymeven	10.97	9.96
sort_g	11.44	6.07
fft16	13.14	11.6
sobel	14.24	13.25
imagerapid	14.36	12.62
cascade	15.45	15.42
GEOMEAN	**11.2112**	**10.29194**

following parameters:

- A_L – The area of the smallest architecture required to successfully route the netlist using a pipelining-unaware place and route flow.
- A_P – The area of the smallest architecture required to successfully route the netlist using a pipelining-aware place and route flow.
- PIPE-COST – The ratio A_P / A_L. This is a quantitative measure of the overhead incurred.

Fig. 22 shows a plot of PIPE-COST vs. the fraction of pipelined signals in a netlist. The eight data points represent the PIPE-COST of each netlist in the benchmark set. It can be seen that an increase in the percentage of pipelined signals in a netlist tends to result in an increase in the PIPE-COST of that netlist. This observation validates our intuition that the fraction of pipelined signals in a netlist roughly tracks the combined architecture and CAD effort needed to successfully route the netlist.

XIII. CONCLUSIONS

The main focus of this work was the development of an algorithm that routes logically retimed netlists on pipelined FPGA architectures. We developed an optimal 1_D router, and

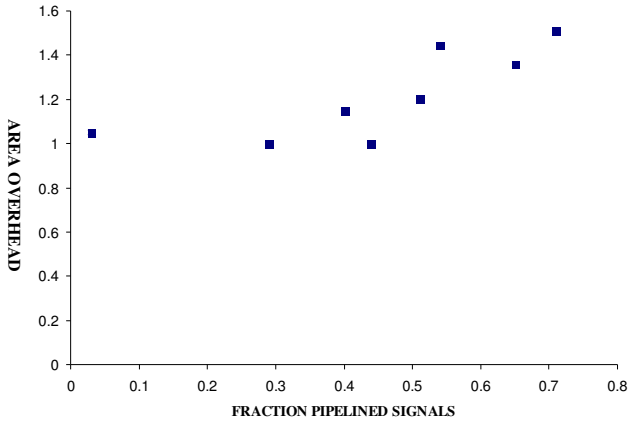

Fig. 22. Experiment 3 – The variation of PIPE-COST vs. fraction pipelined signals.

used it in formulating an efficient heuristic to route two-terminal N_D pipelined signals. The algorithm for routing general multi-terminal pipelined signals borrowed from both the 1_D and N_D routers. Congestion resolution while routing pipelined signals was achieved using Pathfinder. Our results showed that the architecture overhead (PIPE-COST) incurred in routing netlists on the RaPiD architecture was 18%, and that there might be a correlation between the PIPE-COST of a netlist and the percentage of pipelined signals in that netlist.

The formulation of the pipelined routing problem, and the development of the PipeRoute algorithm, proceeded independently of specific FPGA architectures. In the quest for providing programmable, high-throughput architectures, we feel that the FPGA community is going to push towards heavily retimed application netlists and pipelined architectures. When pipelined architectures do become commonplace, PipeRoute would be a good candidate for routing retimed netlists on such architectures.

APPENDIX

A proof that the two-terminal N_D Routing Problem (abbreviated here as $2TN_D$) is NP-Complete via a reduction from the Traveling-Salesman Problem with Triangle Inequality (abbreviated here as TSP-TI):

Traveling-Salesman Problem with Triangle Inequality: *Let G = (V,E) be a complete, undirected graph that has a nonnegative integer cost c(u,v) associated with each edge (u,v)∈E. We must find a tour of G with minimum cost. Furthermore, we have the triangle inequality, that states for all vertices u,v,w∈V, c(u,w) ≤ c(u,v) + c(v,w).*

We consider only problems where |V|>2, since all other cases are trivially solvable. To simplify things, we will convert the original problem to one with strictly positive costs by adding one to each edge cost. Since all solutions to the original problem go through exactly |V| edges, with a solution cost of say 'C', all solutions to the new problem will also have |V| edges, a cost of C+|V|, and correspond exactly to a solution in the original problem. Thus, this transformation is allowable. Note that the triangle inequality holds in this form as well.

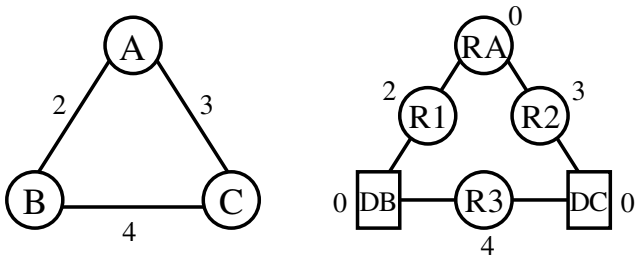

Fig. A1. Example TSP-TI (left) with edge weights, and the corresponding $2TN_D$ (right), with node weights. TSP-TI node A is chosen as the source and the sink, and N = 2.

As stated in [2], TSP-TI is NP-Complete. We can reduce TSP-TI to $2TN_D$ by transforming all TSP-TI nodes to D-nodes, and converting the edge-weights of TSP-TI to R-nodes (Fig. A1). Specifically, let $G_{TSP}=(V_{TSP},E_{TSP})$ be the input graph to TSP-TI, and $G_{2TND}=(V_{2TND},E_{2TND})$ be the corresponding graph we construct to solve TSP-TI with $2TN_D$. Let S_{TSP} be an arbitrary node in V_{TSP}. For each node $M_{TSP}\in V_{TSP}$, create a corresponding node M_{2TND} in V_{2TND}, with cost 0. This node is an R-node if $M_{TSP}=S_{TSP}$, and a D-node otherwise. For each edge $(u,v) \in E_{TSP}$, let x and y be the nodes in V_{2TND} that correspond to u and v respectively. Create a new R-node z in V_{2TND} with cost c(u,v). Also, create edges (x,z) and (z,y) in E_{2TND}. Solve $2TN_D$ with $N=|V_{TSP}|-1$, and S & K = S_{2TND}, the node corresponding to S_{TSP}.

We must now show that the solution to the $2TN_D$ problem gives us a solution to the TSP-TI problem. One concern is that the $2TN_D$ solution may visit some nodes multiple times, either from 0-cost nodes or because wandering paths can be as short as more direct paths. For $2TN_D$ problems on the graphs created from TSP-TI problems, we will define *simplified* $2TN_D$ solutions. Specifically, walk the $2TN_D$ solution path from source to sink. The first time a given D-node is encountered on this walk will be called the *primary* occurrence of that node, and all additional encounters will be called *repeat* occurrences. The occurrences of the source and sink node (which are identical), will be considered primary, and all others repeat. We now eliminate all repeat occurrences to create a simplified $2TN_D$. Specifically, let R_{2TND} be any repeat node on the path, and Pre_{2TND} and $Post_{2TND}$ be the first D-node or source node occurrence on the path before and after R_{2TND} respectively. R_{TSP}, Pre_{TSP}, and $Post_{TSP}$ are the nodes in V_{TSP} that correspond to R_{2TND}, Pre_{2TND}, and $Post_{2TND}$. The cost of the path segment from Pre_{2TND} to $Post_{2TND}$ is equal to the cost of the two R-nodes on this path (since D-nodes and source nodes have a cost of 0), which is equal to $c(Pre_{TSP},R_{TSP}) + c(R_{TSP},Post_{TSP})$. By the triangle inequality, this is no smaller than $c(Pre_{TSP}, Post_{TSP})$. Thus, without increasing the cost of the path, or reducing the number of different D-nodes visited, we can replace the portion of the path from Pre_{2TND} to $Post_{2TND}$ with the path $Pre_{2TND} \rightarrow Rn_{2TND} \rightarrow Post_{2TND}$, where Rn_{2TND} is the node in E_{2TND} corresponding to $(Pre_{TSP},Post_{TSP})$. By recursively applying this process, we will get a simplified $2TN_D$ solution where each D-node appears at most once. Since

$N=|V_{TSP}|-1$ is equal to the number of D-nodes in V_{2TND}, this means that the path visited each D-node exactly once. It also only visits the source node S_{TSP} at the beginning and end of the path. Finally, the cost of the path is no greater than the cost of the original $2TN_D$ solution.

The simplified $2TN_D$ solutions turn out to be solutions for TSP-TI, with the same cost. We can show this by showing that the D-nodes traversed in the $2TN_D$ solution, plus the S_{2TND} node, are a *tour* in TSP-TI. A tour is a simple cycle visiting all nodes in a graph exactly once. In our simplified $2TN_D$ solution all D-nodes are visited exactly once. By converting the path that starts and ends at S_{2TND} into a cycle by fusing together the ends, you also visit S_{2TND} exactly once. The cost of the simplified $2TN_D$ solution is equal to the cost of the R-nodes traversed, which is equal to the cost of the edges between the consecutive vertices in the tour of TSP-TI.

It also turns out that every solution to TSP-TI has an equivalent simplified $2TN_D$ solution with the same cost. Specifically, the tour in TSP-TI can be split at the S_{TSP} node, thus forming a path. The nodes in TSP-TI corresponding to the edges and vertices in the TSP-TI solution constitute a path going through at least $|V_{TSP}|-1$=the number of D-nodes in V_{2TND}, and thus fulfill most of the requirements of $2TN_D$. The only issue to worry about is the restriction in TSP-TI that you cannot enter and exit a D-node on the same edge. However, if $|V_{TSP}| > 2$, then the vertices surrounding a vertex in the TSP-TI path cannot be the same. Thus, TSP-TI never uses the same edge to enter and leave a node, so the equivalent $2TN_D$ solution will never violate the entry/exit rule of $2TN_D$. Again, the cost of the TSP-TI and $2TN_D$ solutions are the same, since the edge weights of TSP-TI are identical to the node weights encountered in the $2TN_D$ solution.

As we have shown, all solutions of TSP-TI have a corresponding, equal cost solution in $2TN_D$, and all simplified $2TN_D$ solutions have corresponding, equal cost solution in TSP-TI. It is also easy to see that there is a polynomial-time method for transforming TSP-TI into $2TN_D$, then map the results of $2TN_D$ to a simplified $2TN_D$ result, and finally convert this into a solution to TSP-TI. Thus, since TSP-TI is NP-Complete, $2TN_D$ is NP-hard.

It is also clear that we can check in polynomial time whether N distinct D-nodes are visited, that the solution is a path starting and ending at S and K respectively, and whether we ever enter and leave a D-node on the same edge. We can also check whether the path length is minimum via binary search on a version requiring path lengths ≤ L. Thus, $2TN_D$ is in NP. Since it is also NP-Hard, $2TN_D$ is therefore NP-Complete.

REFERENCES

[1] Altera Inc., "Stratix™ device family features", available at http://www.altera.com.

[2] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA:1990.

[3] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling, "Architecture design of reconfigurable pipelined datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI,* pp 23-40, 1999.

[4] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath", *6th International Workshop on Field-Programmable Logic and Applications,* pp 126-135, 1996.

[5] S. Kirkpatrick, C. Gelatt Jr., and M. Vecchi, "Optimization by simulated annealing", *Science*, 220, pp. 671-680, 1983.

[6] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995.

[7] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston, MA: 1988.

[8] U. Seidl, K. Eckl, and F. Johannes, "Performance-directed retiming for FPGAs using post-placement delay information", *Design Automation and Test in Europe*, pp. 770 – 775, 2003.

[9] A. Sharma, "Development of a place and route tool for the RaPiD architecture", *Master's Project, University of Washington*, December 2001.

[10] A. Sharma, C. Ebeling, and S. Hauck, "PipeRoute: a pipelining-aware router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 68-77, 2003.

[11] A. Sharma, C. Ebeling, and S. Hauck, "PipeRoute: a pipelining-aware router for FPGAs", *University of Washington, Dept. of EE Technical Report UWEETR-0018*, 2002.

[12] A. Sharma, K. Compton, C. Ebeling, and S. Hauck, "Exploration of pipelined FPGA interconnect structures", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 13-22, 2004.

[13] A. Singh, A. Mukherjee, and M. Sadowska, "Interconnect pipelining in a throughput–intensive FPGA architecture", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp 153-160, 2001.

[14] D. Singh, and S. Brown, "The case for registered routing switches in Field Programmable Gate Arrays", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 161-169, 2001.

[15] D. Singh, and S. Brown, "Integrated retiming and placement for Field Programmable Gate Arrays", *ACM/SIGDA International Symposium on* Field-*Programmable Gate Arrays*, pp 67-76, 2002.

[16] 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", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.

[17] N. Weaver, J. Hauser, and J. Wawrzynek, "The SFRA: a corner-turn FPGA architecture", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3 – 12, 2004.

[18] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement C-slow retiming for the Xilinx Virtex FPGA", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 185 – 194, 2003.

[19] Xilinx Inc., "VirtexII™ platform FPGA features", available at http://www.xilinx.com.

Akshay Sharma received the B.E. degree in Electronics and Communications Engineering from the University of Delhi, New Delhi, India in 1999. He received the M.S. degree in Electrical Engineering from the University of Washington, Seattle, WA in 2001, where he is currently pursuing a Ph.D. in Electrical Engineering. His research interests include VLSI CAD algorithms, high-performance FPGA architectures, and nanocomputing.

Carl Ebeling received the B.S degree in Physics from Wheaton College in 1971 and the Ph.D. degree in Computer Science from Carnegie-Mellon University in 1986. He then joined the Department of Computer Science at the University of Washington, where he is currently Professor of Computer Science and Engineering. He has worked on special-purpose VLSI architectures and CAD tools, including the Triptych FPGA architecture, the Pathfinder routing algorithm and the RaPiD coarse-grained configurable architecture. His current research is focused on programming and compiling for coarse-grained configurable architectures. In a former life, he wrote the Gemini layout-to-schematic comparison program and designed the Hitech chess machine.

Scott Hauck is an Associate Professor of Electrical Engineering at the University of Washington. He received the B.S. in Computer Science from U. C. Berkeley in 1990, and the M.S. and Ph.D. degrees from the University of Washington Department of Computer Science. From 1995-1999 he was an Assistant Professor at Northwestern University. Dr. Hauck's research concentrates on FPGAs, including architectures, applications, and CAD tools. For these efforts he has received an NSF Career Award, a Sloan Fellowship, and a TVLSI Best Paper award.