PipeRoute: A Pipelining-Aware Router for FPGAs

Akshay Sharma
Dept. of Electrical Engineering
University of Washington
Seattle, WA
akshay@ee.washington.edu

Carl Ebeling
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA
ebeling@cs.washington.edu

Scott Hauck
Dept. of Electrical Engineering
University of Washington
Seattle, WA
hauck@ee.washington.edu

ABSTRACT

In this paper we present a pipelining-aware router for FPGAs. The problem of routing pipelined signals is different from the conventional FPGA routing problem. For example, the two terminal N-Delay pipelined routing problem is to find the lowest cost route between a source and sink that goes through at least N (N > 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 the delay constraint at *each* sink is satisfied.

We begin this work by proving that the two terminal N-Delay problem is NP-Complete. We then propose an optimal algorithm for finding a lowest cost 1-Delay route. Next, the optimal 1-Delay router is used as the building block for a greedy two terminal N-Delay router. Finally, a multi-terminal routing algorithm (PipeRoute) that effectively leverages the 1-Delay and N-Delay routers is proposed.

We evaluate PipeRoute's performance by routing a set of retimed benchmarks on the RaPiD [3] architecture. Our results show that the architecture overhead incurred in routing retimed netlists on RaPiD is less than a factor of two. Further, the results indicate a possible trend between the architecture overhead and the percentage of pipelined signals in a netlist.

1. INTRODUCTION

It is well established that FPGAs are a convenient marriage between the flexibility of software, and performance levels achievable in hardware. Reconfigurable logic units, coupled with a rich programmable interconnect structure, can be used to implement a variety of applications. However, while FPGAs remain extremely attractive for their hardware flexibility, the minimum clock period that is achievable in present-day FPGAs leaves a lot to be desired.

In the world of microprocessors and custom design, pipelining is widely used to reduce the critical path delay of a circuit. The development of powerful sequential retiming heuristics has contributed to reducing the clock period of circuits even further [5,6]. Thus, designers of reconfigurable architectures are now paying serious attention to providing pipelining resources in the logic units and routing fabric that constitute reconfigurable architectures.

A number of research groups have proposed pipelined FPGA architectures. HSRA [13] is an example of an FPGA architecture that has a hierarchical, pipelined interconnect structure. A fraction of the switchboxes is populated with registered switches to meet a target clock period. Also, instead of having a single register on the output of a LUT (which is generally the case in existing FPGA architectures), a bank of registers is connected to each input of the LUT. This helps balance path delays introduced by the pipelined interconnect. User applications are mapped to HSRA by integrating data retiming with a conventional FPGA CAD flow.

A second example of a pipelined FPGA architecture is proposed in Singh et al [10]. The routing architecture is hierarchical, and the higher-level routing consists of horizontal and vertical long lines that surround logic blocks. Each long line is pipelined using a bank of registered switch-points, and every switch-point can be used to delay a long line from 0 – 4 clock cycles. DSP designs mapped to this architecture were able to achieve throughputs of up to 600 MHz.

RaPiD [3,4] is a coarse-grained one-dimensional (1-D) architecture that has pipelined datapath and interconnect structures. The datapath consists of 16-bit ALUs, multipliers, SRAMs and registers. The registers comprise a significant fraction of the datapath, thus providing pipelining resources. The interconnect is composed of short tracks that are used to achieve local communication between logic units, and long tracks that enable relatively long distance communication along the datapath. The long tracks traverse multiple switch-points, whereas the short tracks do not traverse any switch-points. The outputs of every logic unit, as well as all switch-points, can optionally be registered. Due to the 1-D nature of the interconnect, switchpoints have 2 terminals, and are bidirectional. Like the architecture proposed in [10], the RaPiD architecture is targeted at regular, compute intensive applications that are amenable to deep pipelining.

The aforementioned architectural examples indicate that good progress is being made in the design of pipelined architectures. The challenge now is to develop CAD tools that can map user applications to pipelined FPGA

architectures. In [12], the authors investigate the benefits of integrating placement and retiming by proposing retiming aware placement algorithms. The same authors present a retiming aware router in [11]. This router attempts to place long signals on tracks that have registered switches, so that a subsequent retiming step can take advantage of the assignment to pipeline the long signals. In [11], the goal is to reduce interconnect delay by pipelining long signals. Placement and logic retiming are closely coupled to give the retiming step an estimate of routing delay in [12].

The subject of this paper is the development of an algorithm called PipeRoute that routes *retimed* application netlists on *pipelined* FPGA architectures. In *retimed* netlists, all pipelining registers are explicitly enumerated, and it is therefore possible to calculate the number of clock cycles that separate the signal's source from each of its sinks. A *pipelined* FPGA architecture is one that has pipelining resources in the *interconnect structure*. These pipelining resources supplement the registers that are already provided in FPGA logic blocks. PipeRoute takes a retimed 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 routing algorithm that is capable of routing retimed 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 abstractly represented as a graph consisting of routing- and pipelining-nodes.

2. PROBLEM BACKGROUND

The FPGA routing problem is to determine an assignment of signals to limited routing resources while trying to achieve the best possible delay characteristics. Pathfinder [7] is one of the most widely used FPGA routing algorithm. 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 the 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.

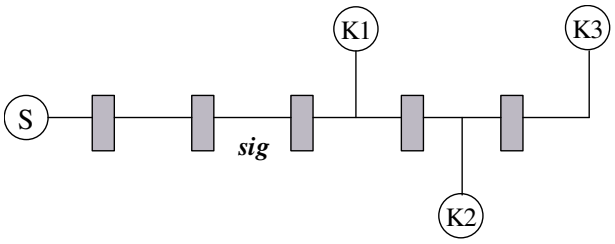

Fig. 1: A multi-terminal pipelined signal

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 delay constraints at every sink. Due to the fact that there are a fixed number of sites in the interconnect where a signal can be delayed by a clock cycle (hereafter referred to as "delay sites"), it can be easily seen that a route that is found for ***sig*** by a conventional, pipelining-unaware FPGA router may not contain enough delay sites to satisfy the delay 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 can be stated as:

Two-terminal N-Delay 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* $S,K \in V$ *be two R-nodes. Find a path* $P_G(S,K)$ *that connects nodes S and K, and contains at least N (*$N \geq 1$*) distinct D-nodes, such that* $w(P_G(S,K))$ *is minimum, where*

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

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-Delay" route. R-nodes represent wires and IO pins of logic units in a pipelined architecture, whereas D-nodes represent registered switch-points. A registered switch-point 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. Under this framework, an abstraction of 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 ($N \geq 1$) distinct D-nodes (N is the number of clock cycles that separates the source from the sink). Note that a lowest cost route can be self-intersecting i.e. R-nodes can be shared in the lowest cost route. We have shown that the two-terminal N-Delay problem is NP-Complete (Appendix A). In the more complex case of a multi-terminal signal, the problem is to find an MST that contains enough D-nodes such that *each* sink is the correct number of clock cycles away from the source.

A simple solution to the pipelined routing problem would be to address pipelining in the placement phase. The pipelining registers in a netlist could be mapped to registered switch-points in the architecture, and a simulated annealing placement algorithm could determine an optimal placement of the pipelining registers. After the placement phase, a conventional FPGA router could be used to route the signals in the netlist. However, a placement of a netlist that 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. It is therefore clear that a pipelining-aware placement phase is not sufficient to successfully route pipelined signals.

In sections 3, 4 and 5, we present a greedy heuristic search algorithm for routing signals on pipelined FPGA architectures, and an explanation of how we use Pathfinder's Negotiated Congestion (NC) algorithm [7] in conjunction with our heuristic to resolve congestion. Section 6 describes the target architecture that we used in our experiments, while Section 7 describes the placement algorithm we developed to enable our routing approach. We describe our experimental setup and test strategy in Section 8, followed by results in Section 9. Finally, in Section 10, we discuss some of many directions for future efforts, and conclude this work.

3. ONE-DELAY 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. However, we now show that a lowest cost route between a source and sink that goes through at least 1 D-node can be found in polynomial time. In a weighted, undirected graph, the Breadth First Search (BFS) algorithm is widely used to find the lowest cost route between a source and sink. The remainder of this section evaluates several modifications of conventional BFS that can be used to find a lowest cost 1-Delay route. Our first modification is *Redundant-Phased-BFS*. 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 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-BFS*. 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*-BFS is F times that of conventional BFS. Since F could potentially equal the number of registers in the FPGA, the worst-case run-time of *Redundant-Phased-BFS* could get prohibitive.

An alternative to *Redundant-Phased-BFS* that can be used to find a lowest cost 1-Delay route between a source and sink is *Combined-Phased-BFS*. This algorithm attempts to reduce run-time by combining the search space of all the D-nodes that can be explored at phase 0 from the source. The only difference between *Redundant-Phased-BFS* and *Combined-Phased-BFS* 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-BFS* is only double that of conventional BFS. In addition, an important effect of the dichotomy that we have created due to phase 0 and phase 1 wavefronts is that R-nodes that constitute the phase 0 segment of a 1-Delay route can be reused in the phase 1 segment of the same 1-Delay route. We rely on Pathfinder's [7] congestion resolution scheme to adjust the history cost of such R-nodes, so that in a later iteration a 1-Delay route with no node reuse between phase 0 and phase 1 segments can be found.

A step-by-step illustration of how *Combined-Phased-BFS* works is shown in Figs. 2A through 2E. 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 BFS. The number 0 next to S in Fig. 2A indicates that S has been explored by a phase 0 wavefront. In Fig. 2B, the neighbors of S are explored by the phase 0 wavefront initiated at S. The 2^{nd}-level neighbors of S are explored by phase 0 in Fig. 2C, 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. 2D, the neighbors of D1 (excluding R1) are explored at phase 1. The phase 0 exploration also continues simultaneously, and note how nodes R4 and R7 have been explored by both phase 0 and phase 1 wavefronts. Finally, in Fig. 2E, the sink K is explored by the phase 1 wavefront initiated at D1. The route found by *Combined-Phased-BFS* is shown in boldface in Fig. 2E, and is in fact an optimal route between S and K.

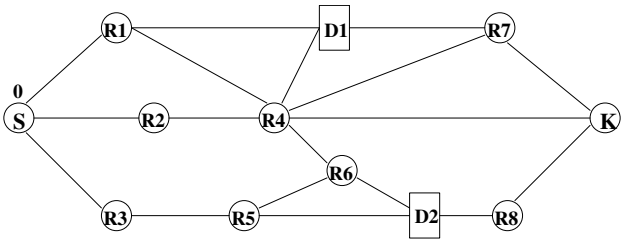

Fig. 2A: Phase 0 exploration commences at node S.

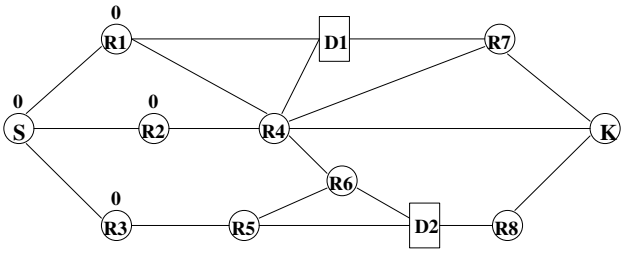

Fig. 2B: The neighbors of S are explored at phase 0.

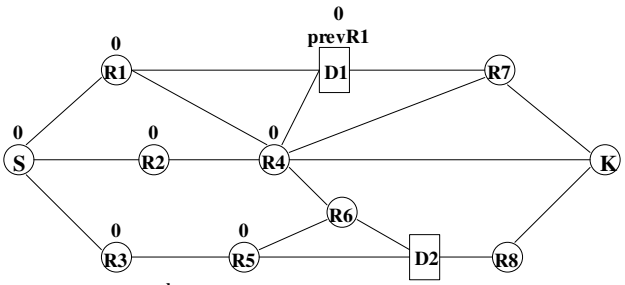

Fig. 2C: 2nd-level neighbors of S are explored at phase 0, and in the process D-node D1 is discovered.

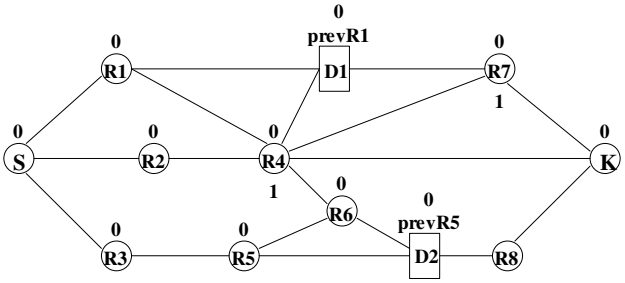

Fig. 2D: D1 starts a phase 1 exploration. The phase 0 exploration continues simultaneously, and D2 is discovered.

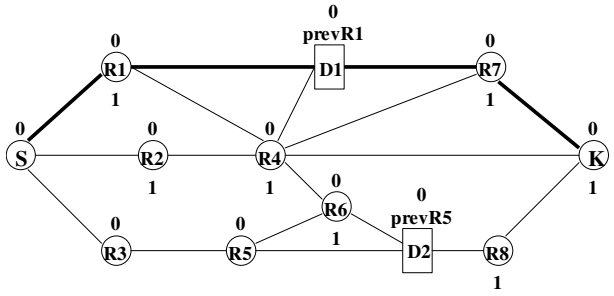

Fig. 2E: K is explored by phase 1 wavefront commenced at D1.

Unfortunately, *Combined-Phased-BFS* 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-BFS* 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-BFS* 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-BFS* 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 delay site. 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-BFS* 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.

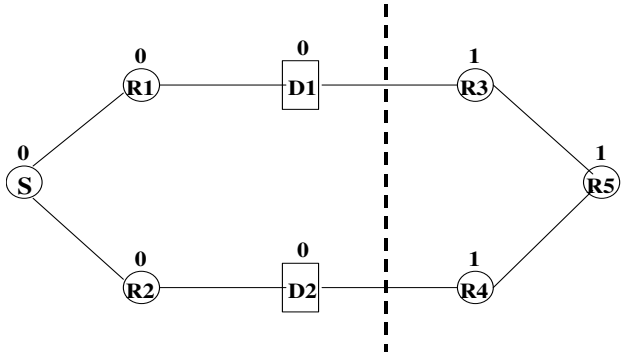

Fig. 3: A case for which phased BFS fails. Observe how the phase 1 exploration has got isolated from the phase 0 exploration

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-Delay route to K from being found. To address the nature of transitions from phase 0 to phase 1, 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-Delay path S-R2-R3-R4-D1-R1-K to be found.

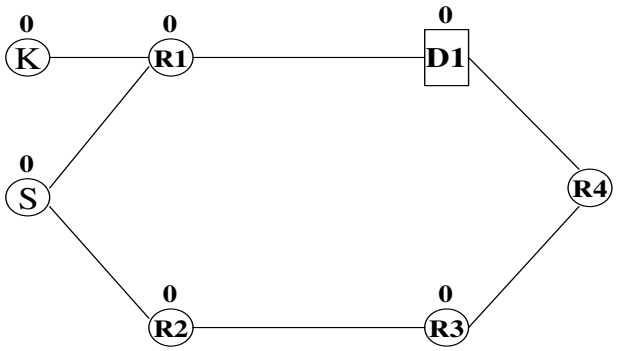

Fig. 4: D1 is explored at phase 0 from R1, thus precluding the discovery of the 1-Delay path to the sink K.

The following rules summarize *2Combined-Phased-BFS*:

- An R-node can be explored at most once at phase 0.
- A D-node can be explored at most twice at phase 0.
- An R-node can be explored by at most two distinct phase 1 explorations. The cases in which two phase 1 explorations are distinct are:
    - The two phase 1 explorations are initiated by two different D-nodes, OR
    - The two phase 1 explorations are initiated by the same D-node, but the R-nodes that were used to explore the D-node at phase 0 are different.
- A D-node can be explored by at most two distinct phase 1 explorations. This rule is identical to the way R-nodes are explored at phase 1.

We have proven that *2Combined-Phased-BFS* finds an optimal 1-Delay route between a source and sink on an undirected graph consisting of R-nodes and D-nodes. A detailed proof can be found in Appendix B.

4. N-DELAY ROUTER

In this section, we present a heuristic that uses the optimal 1-Delay router to build a route for a two terminal N-Delay signal. This heuristic greedily accumulates delay at the sink by using 1-Delay routes as building blocks. In general, an N-Delay route is recursively built from an (N-1)-Delay route by successively replacing each segment of the (N-1)-Delay route by a 1-Delay route and then selecting the lowest cost N-Delay route. Fig. 5 is an abstract illustration of how a 3-Delay route between S and K is found. In the first step, we find a 1-Delay route between S and K, with D11 being the D-node where we pick up delay. At this point, we increment the sharing cost of all nodes that constitute the route S-D11-K. In the second step, we find two 1-Delay 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-Delay route between S and D11 (S-D21-D11). Store cost of route S-D21-D11-K in $Cost_{S\text{-}D21\text{-}D11\text{-}K}$.
- Restore segment S-D11 by incrementing the sharing cost of segment S-D11.
- Decrement sharing cost of segment D11-K.
- Find 1-Delay route between D11 and K (D11-D22-K). Store cost of route S-D11-D22-K in $Cost_{S\text{-}D11\text{-}D22\text{-}K}$.
- Restore segment D11-K by incrementing the sharing cost of segment D11-K.
- Select the lowest cost route, either S-D21-D11-K and S-D11-D22-K.

Suppose the lowest cost 1-Delay 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-Delay segments, and for each case the cost of the entire 3-Delay route between S and K is stored. The lowest cost route is then selected. In Fig. 5, the 3-Delay route that is found is shown in dark lines, and is S-D11-D31-D22-K.

The number of 1-Delay BFS' launched for the 3-Delay route that we just discussed is 1 + 2 + 3 = 6. For the general N-Delay case, the number of 1-Delay BFS' launched is 1 + 2 + ... + N = N(N-1)/2. A bound on the number of 1-Delay BFS' launched for an N-Delay route is N^2.

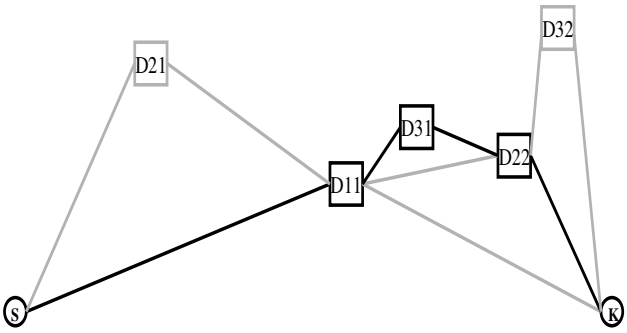

Fig. 5: Building a 3-Delay route from 1-Delay routes

5. MULTI-TERMINAL ROUTER

The previous section described a heuristic that uses optimal 1-Delay routes to build a two-terminal N-Delay

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 delays 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 delays respectively. We will now demonstrate how a route for a multi-terminal signal can be found by taking advantage of the 1-Delay and N-Delay routers that were discussed in Sections 3 and 4.

The routing tree for a multi-terminal pipelined signal is built one sink at a time. The entire list of sinks is stored in a pre-sorted list called *sorted_sink_list*, and each sink is considered in non-decreasing order of delay separation from the source of the signal. Hence, the multi-terminal router starts by finding a route to a sink that is the least number of delays away from the source. Since finding a route to the first sink is a two-terminal case, we use the two-terminal N-Delay 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.

We explain the multi-terminal router via a simple example. Assume a hypothetical signal that has a source S and sinks K3 and K4. K3 must be separated from S by 3 delays, whereas K4 must be separated by 4 delays. Sink K3 is considered first, and the N-Delay router is used to find a 3-Delay route between S and K3. In Fig. 6A, the route S-D1-D2-D3-K3 represents the 3-Delay route between S and K3, 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.

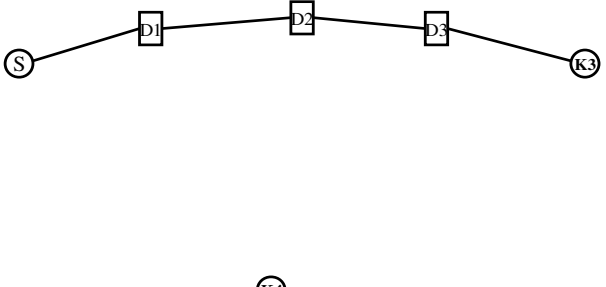

Fig. 6A: 3-Delay route to K3 using the two-terminal N-Delay router. S-D1-D2-D3 is the *partial_routing_tree*.

After a route to K3 is found, the router considers sink K4. As was the case in the N-Delay router, we accumulate delay at K4 one delay at a time. Thus, we start by finding a 1-Delay route to K4, then a 2-Delay route, a 3-Delay route, and finally a 4-Delay route to K4. It can be seen that a 1-Delay route to K4 can be found either from the 0-Delay segment S-D1 by going through another D-node, or from the 1-Delay 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-Delay BFS 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-Delay BFS in such a way could find a 1-Delay path from S-D1 or a 0-delay path from D1-D2, depending on which is of lower cost. Assume that the route to K4 that is found is the gray segment P-K4 in Fig. 6B. Once the segment P-K4 is found, the sharing cost of the nodes that constitute P-K4 is incremented. The segment P-K4 is called the *surviving_candidate_tree*. The *surviving_candidate_tree* can be defined as the tree that connects the sink (K4 in this case) under consideration to some node in the *partial_routing_tree* every time an N-Delay route ($1 \le N \le 4$ in this case) to the sink is found. Thus, a distinct *surviving_candidate_tree* results immediately after finding the 1-Delay, 2-Delay, 3-Delay and 4-Delay routes to K4.

Next, we attempt to find a 2-Delay route to K4. Before explaining specifics, it is important to point out here that while finding an N-Delay route to a sink in general we try two options. The first is to use the N-Delay and (N-1)-Delay segments in the *partial_routing_tree* together to start a 1-Delay BFS. The other option is to alter the *surviving_candidate_tree* to include an additional D-node as was done in the two terminal N-Delay router. The lower cost option is chosen, and this option becomes the new *surviving_candidate_tree*.

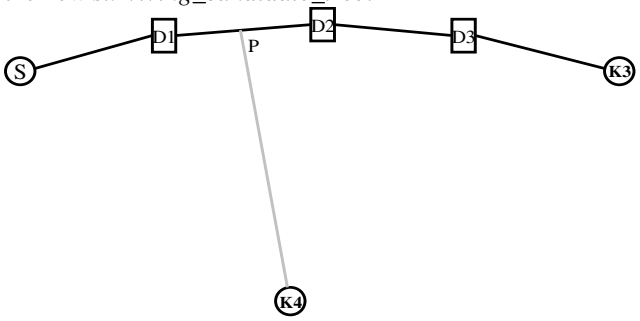

Fig. 6B: 1-Delay route to K4. P-K4 is found by launching a 1-Delay BFS that starts with segment S-D1 at phase 0 and segment D1-D2 at phase 1. P-K4 is the *surviving_candidate_tree.*

Thus, for finding a 2-Delay route to K4, we first launch a 1-Delay BFS using segments D1-D2 and D2-D3 and store the cost of the route that is found. Then, we rip up segment P-K4 (Fig. 6B) and replace it with a 1-Delay route between segment D1-D2 and K4, and store the cost of the 1-Delay route. The lower cost route is selected, and the sharing cost of the nodes that constitute this route is incremented. This selected route becomes the new *surviving_candidate_tree*. In Fig. 6C, assume that the lower cost route that is selected is the segment P1-Da-K4 shown in gray.

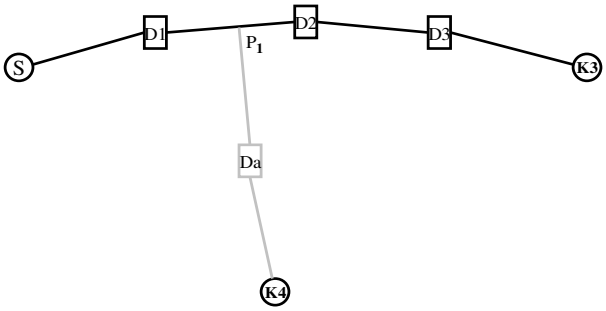

Fig. 6C: 2-Delay route to K4. P1-Da-K4 is now the *surviving_candidate_tree.*

A similar reasoning can be applied to finding a 3-Delay route to K4. A 1-Delay BFS using segments D2-D3 and D3-K3 (which are shown at delay 2 and 3 respectively in Fig. 6D) is launched, and the cost of the resulting route is stored. Then, the *surviving_candidate_tree* P1-Da-K4 (Fig. 6C) is modified to add another D-node much in the same manner that a two-terminal 2-Delay route is built from an already established 1-Delay route (Section 4). The cost of the modified *surviving_candidate_tree* is also stored. The lower cost route is selected, and the sharing cost of relevant nodes incremented. In Fig. 6D, assume that the lower cost route that is selected is P1-Da-Db-K4. This route now becomes the *surviving_candidate_tree*.

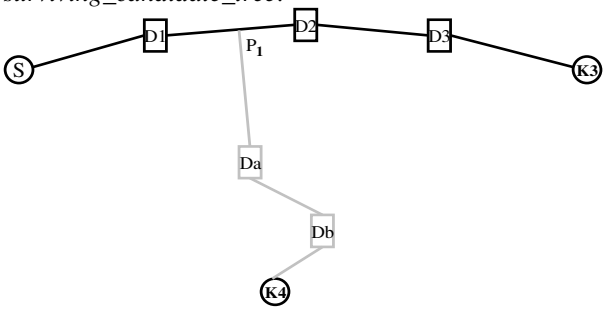

Fig. 6D: 3-Delay route to K4. P1-Da-Db-K4 is the resulting *surviving_candidate_tree.*

Finally, in Fig. 6E, the cost of finding a 1-Delay route to K4 from the segment D3-K3 (which is at delay 3) proves to be less than the cost of the route that modifies the *surviving_candidate_tree* P1-Da-Db-K4 (Fig. 6D). The segment P1-Da-Db-K4 is ripped up, and the segment P3-D4-K4 is joined to the *partial_routing_tree* to complete the routing to K4.

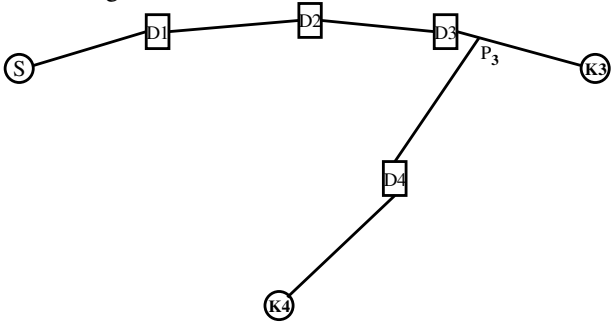

Fig. 6E: 4-Delay route to K4. P3-D4-K4 is the final *surviving_candidate_tree*, and this tree is joined to the *partial_routing_tree* to complete routing to K4.

6. TARGET ARCHITECTURE

In this section, we briefly describe the features of a simplified RaPiD [3] architecture. The reasons that influenced us to use simplified RaPiD as the target architecture for our experiments are:

- RaPiD has a pipelined datapath structure. More importantly, it provides bi-directional pipelining sites in the interconnect.
- We have easy access to the in-house RaPiD compiler and retimer. Thus, we have been able to generate a representative set of benchmark applications for our experiments.

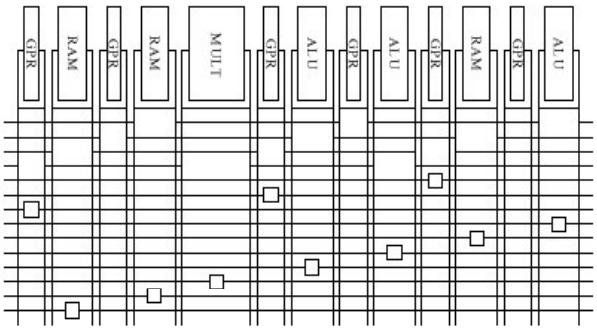

Fig. 7: An example of a RaPiD [3] architecture cell. Several RaPiD cells can be tiled together to create a representative architecture.

The 1-Dimensional (1-D) RaPiD datapath (Fig. 7) consists of coarse-grained logic units that include ALUs, multipliers, small SRAM blocks, and registers. Each logic unit is 16-bit wide. The interconnect consists of 1-D routing tracks that are also 16-bit 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. 7, the uppermost 5 tracks are short tracks, while the remaining tracks are long tracks. Each input of a logic unit can be driven by any routing track by means of a multiplexer. Similarly, the outputs of a logic unit can be configured to drive any routing track. An output can drive multiple routing tracks.

The long tracks in the RaPiD interconnect are segmented by means of Bus-Connectors (shown as empty boxes in Fig. 7 and abbreviated BCs), which are essentially bi-directional delay sites. In the simplified version of RaPiD that we used in our experiments, each BC can be used to pick up either 0 or 1 clock cycle delay. Thus, a BC can be used in transparent (0 clock cycle delay) or registered (1 clock cycle delay) mode. Another aspect of RaPiD is that datapath registers can be used to switch tracks. At the end of the placement phase, all unoccupied datapath

registers are included in the routing graph as unregistered switch-points. The ability to switch tracks provides an important degree of flexibility while attempting to route netlists on RaPiD.

7. PLACEMENT

The two inputs to the placement program are descriptions of a retimed RaPiD netlist and the target RaPiD architecture. The RaPiD compiler generates application netlists in an internal format, and the architecture is represented as an annotated structural Verilog file. For the sake of nomenclature, the logical components that constitute the netlist will be referred to as "instances" from this point onwards. The final placement of the netlist is determined using a Simulated Annealing [8] algorithm. A good cooling schedule is essential to obtain high-quality solutions in a reasonable computation time with simulated annealing. For our placement program, we used the cooling schedule developed for the VPR tool-suite [1].

The development of a representative cost-function for the placement program is an interesting problem. Since the number of routing tracks in the interconnect fabric of the RaPiD architecture is fixed, we capture the quality of the placement by means of a *cutsize* metric. The *cutsize* at a vertical partition of the architecture is defined as the number of signals that need to be routed across that partition for a given placement of the netlist. The *max_cutsize* is defined as the maximum *cutsize* that occurs at any vertical partition of the architecture. The *total_cutsize* is defined as:

$$total\_cutsize = \sum_{j=1}^{j=Y} (cut\_size)_j$$

where Y is the total number of logic resources that constitute the architecture. The *avg_cutsize* is then defined as:

$$avg\_cutsize = total\_cutsize/Y$$

Both *max_cutsize* and *avg_cutsize* are important estimates of the routability of a netlist. Since the RaPiD architecture provides a fixed number of tracks for routing signals, it is necessary to formulate a placement cost function that favorably recognizes a move that decreases *max_cutsize*. At the same time, it is clear that a simple cost function that attempts to reduce only *max_cutsize* will be inadequate. A cost function that is determined only by *max_cutsize* will not be able recognize changes in *avg_cutsize*. This means that the annealer will accept moves that increase *avg_cutsize*, but do not change *max_cutsize*. Such zero-cost moves may cumulatively increase the overall congestion in the datapath considerably, thus making it harder for the annealer to find the sequence of moves that will reduce *max_cutsize*. It can thus be concluded that *avg-cutsize* should also contribute to the cost of a placement. Reducing *avg_cutsize* not only reduces overall congestion in the datapath, but also lowers the total wire-length. The cost function is therefore formulated as follows:

$$cost = w*max\_cutsize + (1-w)*avg\_cutsize$$

where $0 \le w \le 1$. The value of w was empirically determined to be 0.3. A detailed discussion of how the value of w was determined can be found in [9].

The development of the placement approach so far has focused only on the reduction of track count and wirelength, and this approach works well in conjunction with a pipelining-unaware router that attempts only connectivity routing [9]. As a next step, we need to include pipelining information in our placement cost function so that the router can find pipelined routes in an effective manner. Recall from Section 1 that a retimed RaPiD netlist explicitly enumerates all pipelining registers. At the same time, the architecture file contains information about the location and connectivity of every delay site (BCs) in the architecture. Since we have prior knowledge of the location and track connectivity of the BCs that are provided by the architecture, we simply map each pipelining register in the netlist to a unique physical BC in the architecture. The placement program's move function is modified to include pipelining registers during simulated annealing. Our high-level objective in mapping pipelining registers to BCs is to place netlist instances such that the router is able to find sufficient delay resources while routing pipelined signals.

The calculation of cutsize contributions due to pipelining registers is markedly different from the calculation of cutsize contributions due to other types of netlist instances. This is because of an important difference between the connectivity of BCs and the connectivity of datapath logic resources. Both terminals of a BC directly connect to adjacent segments of the same routing track (Fig. 7), whereas the input and output terminals of all datapath logic resources can connect to any routing track. Thus, if two instances are mapped to ALU positions X1 and X2 in the architecture, the cutsize contribution due to a two-terminal signal that connects the two instances is simply |X1-X2|. However, the same reasoning cannot be directly extended to pipelining registers that are mapped to BCs. For example, consider a two-terminal signal *sig* that connects pipelining registers mapped to D1 and D2 in Fig. 8. Since D1 and D2 are on separate tracks, the router would have to switch tracks to route *sig*. If the nearest available switch-point in the datapath is at position Xsw (Xsw > X2), then the cutsize contribution due to *sig* is (Xsw – X1) + (Xsw-X2), and not merely (X2-X1). Thus, the cutsize contributions due to signals that connect to pipelining registers are very sensitive to the placement of the pipelining registers, especially if the pipelining registers are mapped to BCs on different tracks. The cutsize contributions due to pipelining

registers are estimated and included in our annealer's cost function.

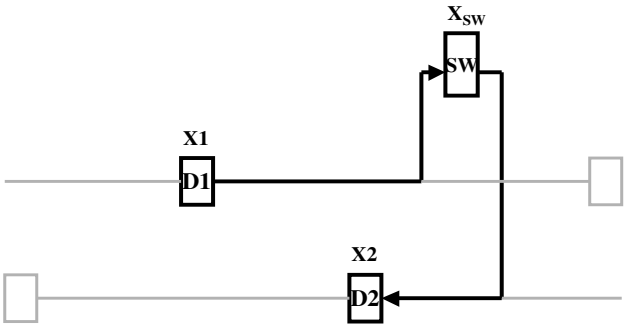

Fig. 8: Calculating the cost of a two-terminal signal that connects D1 and D2. To route this signal, the router would have to switch tracks in the datapath. The cutsize contribution due to thus signal is (Xsw-X1) + (Xsw-X2)

8. TESTING SETUP

The individual placement and routing algorithms that we implemented are as follows:

- SimplePlace – This placement algorithm is pipelining unaware i.e. it attempts to reduce track count and wirelength without taking pipelining into account [9].
- PipePlace – This placement algorithm is derived from SimplePlace and is pipelining aware. It attempts to place netlist instances such that the router is able to find enough delay resources while routing pipelined signals.
- Pathfinder – This routing algorithm is pipelining unaware i.e. it attempts only connectivity routing without considering pipelining information [7].
- PipeRoute – This is the pipelining aware routing algorithm that we presented in Sections 3, 4 and 5.

We measure the quality of combined place-and-route approaches in terms of:

- The size of the architecture needed to route a netlist. The size of an architecture is measured in terms of number of RaPiD cells (Fig. 7).
- The minimum number of routing tracks that we need to route a netlist on a given architecture.

The pipelining-unaware place and route algorithms are included to give us a lower-bound on the size of the architecture and the minimum number of routing tracks needed to place and route retimed netlists.

Test architectures are generated using software provided by Northwestern University graduate student Katherine Compton. This software is capable of generating RaPiD architectures that have a user-specified number of RaPiD cells. Further, it is possible to specify the number of short tracks per cell, long tracks per cell, and bus-connectors per long track per cell. In all test architectures, approximately $2/7^{th}$ of the tracks are short tracks, and $5/7^{th}$ of the tracks are long tracks. Each short track consists of 4 segments per cell, and each long track has 3 BCs per cell.

We use retimed benchmark netlists generated by the RaPiD compiler. The benchmark set consists of three different FIR filter implementations, two implementations of sorting, a 16-point FFT, a matrix multiplier, two different digital camera filters, and a netlist that calculates logarithms using a series expansion. The composition of each benchmark netlist is shown in Table 1. Columns 2 – 6 show the number of 16-bit ALUs, 16x16 multipliers, 256x16 SRAMs, 16-bit data registers, and 16-bit pipelining registers respectively. Column 7 shows the percentage of signals in the netlist that are pipelined.

Netlist	16-bit ALUs	16x16 Mults	256x16 SRAMs	Data Regs	Pipe Regs	% Pipelined
fft16_2nd	24	12	12	29	29	7%
img_filt	47	17	13	85	29	8%
mux_corr	3	6	6	16	6	13%
cascade	8	8	8	24	29	21%
matmult	8	4	12	10	22	23%
firTM	31	16	32	90	149	23%
firsymeven	31	16	0	47	184	36%
sortG	29	0	16	60	175	47%
log8	56	48	0	66	635	47%
sort2DRB	22	0	8	46	128	60%
med_filt	45	1	4	39	241	84%

Table 1: Benchmark composition

9. RESULTS

We present the results of our experiments in this section. We acquired data by running the entire set of benchmarks through two place-and-route approaches. The first approach uses SimplePlace to place the netlist, and then uses Pathfinder to do connectivity routing. This approach treats the benchmarks as if they were unpipelined, and is used as a lower bound. The second approach places netlists using PipePlace, and uses PipeRoute to do pipelined routing. For both approaches, we recorded the size of the smallest RaPiD architecture on which each netlist successfully routed, and the minimum number of routing tracks that were required to route the netlist. We then defined the following result metrics:

- N_{SIM} – The minimum number of RaPiD cells required to route a netlist using pipelining-unaware placement and routing algorithms (SimplePlace and Pathfinder respectively).
- N_{PIPE} – The minimum number of RaPiD cells required to route a netlist using pipelining-aware placement and routing algorithms (PipePlace and PipeRoute).
- T_{SIM} – The minimum number of routing tracks required to route a netlist on an architecture of size N_{SIM} using a pipelining-unaware router (Pathfinder).

- T_{PIPE} – The minimum number of routing tracks required to route a netlist on an architecture of size N_{PIPE} using a pipelining-aware router (PipeRoute).
- A_{XP} – The ratio of N_{PIPE} to N_{SIM}.
- T_{XP} – The ratio of T_{PIPE} to T_{SIM}.
- PIPE-COST – The multiplication of A_{XP} and T_{XP}. This is a quantitative measure of the overhead we incur in trying to place and route retimed netlists on RaPiD architectures.

Table 1 shows the results we obtained. The netlists that constitute the benchmark set are in column 1. Column 2 contains the N_{SIM} value for each netlist. Note that for each netlist in the benchmark set, N_{SIM} was found to be equal to the minimum number of RaPiD cells required to implement the logic of the netlist irrespective of routing requirements. The table is sorted in non-decreasing order of N_{SIM}. Column 3 shows the percentage of signals in each netlist that are pipelined. This percentage is a measure of the pipelining difficulty of a netlist. Column 4 shows the value of A_{XP} for each netlist, while column 5 shows the value of T_{XP}. The PIPE-COST for each netlist is presented in column 6.

From Table 2, we see that the mean architecture expansion overhead due to pipelined routing is 20%, while the mean track expansion overhead is 45%. Overall, the cost of routing retimed netlists is slightly less than double that of routing the same netlists without taking pipelining into account. Fig. 9 is a scatter diagram that plots the PIPE-COST of each netlist in the benchmark set vs. the minimum number of RaPiD cells that were required to fit that netlist. There is evidently no correlation between the size of a netlist and its PIPE-COST. However, a potential trend can be observed in Fig. 10, which plots the PIPE-COST of each netlist vs. the percentage of signals that are pipelined in that netlist. 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 is a promising trend, since it gives us the ability to make a rough estimate of the PIPE-COST of a netlist based on the fraction of pipelined signals in that netlist.

Netlist	N_{SIM}	% Pipelined	A_{XP}	T_{XP}	PIPE-COST
matmult	4	23%	1	1.5	1.5
mux_corr	6	13%	1	1.2	1.2
cascade	8	21%	1	1	1
sort2DRB	8	60%	1.75	1.33	2.33
fft16_2nd	12	7%	1	1.3	1.3
sortG	12	47%	1.67	1.67	2.77
firTM	16	23%	1.25	1.8	2.25
firsymeven	16	36%	1	1.6	1.6
med_filt	16	84%	1.63	1.44	2.35
img_filt	18	8%	1	1.4	1.4
log8	48	47%	1.25	2	2.5
Geometric Mean			**1.2**	**1.45**	**1.74**

Table 2: Variation in PIPE-COST across benchmark set

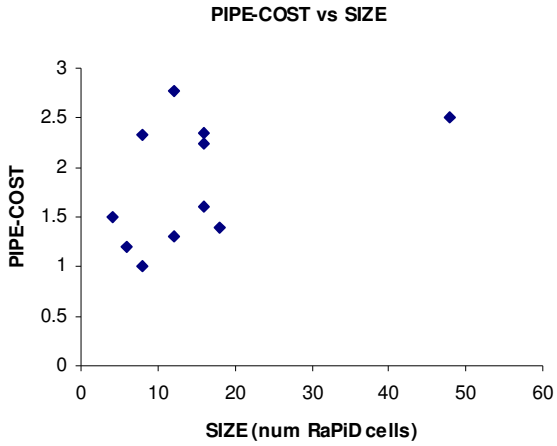

Fig. 9: Variation in PIPE-COST w.r.t size across the benchmark set

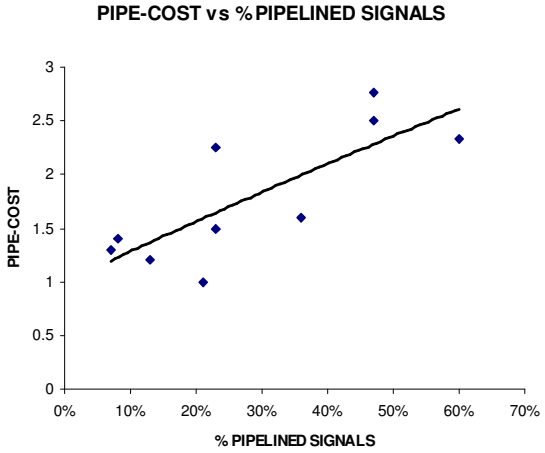

Fig. 10: Variation in PIPE-COST with % pipelined signals across the benchmark set

10. CONCLUSIONS & FUTURE WORK

The main focus of the work described in this paper was the development of an algorithm that routes logically retimed circuits on pipelined FPGA architectures. We developed an optimal 1-Delay router, and used it in formulating an efficient heuristic to route two-terminal N-Delay pipelined signals. The algorithm for routing general multi-terminal pipelined signals borrowed from both the 1-Delay and N-Delay routers. Congestion resolution while routing pipelined signals was achieved using Pathfinder. Our results showed that the architecture overhead (PIPE-COST) of routing logically retimed netlists on the RaPiD architecture was 1.74, and that there is some correlation between the PIPE-COST of a netlist and the percentage of pipelined signals in that netlist.

An important aspect of this work is that 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, the PipeRoute algorithm would be a good candidate for routing retimed netlists on such architectures.

This work has spawned several research vectors that can be actively explored in the future. An important direction is the development of more sophisticated placement algorithms, as a powerful pipelining-aware placement tool might improve the performance of PipeRoute. A second direction for future work lies in the development of pipelined routing algorithms optimized for run-time. Specifically, the search-space of the multi-terminal router could be intelligently reduced to obtain quality solutions in shorter run-times. Finally, PipeRoute could be used in conjunction with an appropriate pipelining-aware placement tool for architecture exploration vis-a-vis numbers and locations of registered switch-points in FPGA interconnect structures.

11. ACKNOWLEDGMENTS

We would like to thank the RaPiD group at the University of Washington for giving us access to the RaPiD compiler. Thanks are also due to Katherine Compton at Northwestern University for providing us the architecture generation program. 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.

REFERENCES:

[1] V. Betz and J. Rose, ‘‘VPR: A New Packing, Placement and Routing Tool for FPGA Research,” *Seventh International Workshop on Field-Programmable Logic and Applications,* pp 213-222, 1997.

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

[3] Darren C. Cronquist, Paul Franklin, Chris Fisher, Miguel Figueroa, and Carl Ebeling. "Architecture Design of Reconfigurable Pipelined Datapaths," *Twentieth Anniversary Conference on Advanced Research in VLSI,* pp 23-40, 1999.

[4] Carl Ebeling, Darren C. Cronquist, Paul Franklin. "RaPiD - Reconfigurable Pipelined Datapath", *6th International Workshop on Field-Programmable Logic and Applications,* pp 126-135, 1996.

[5] C. Leiserson, F. Rose, and J. Saxe, “Optimizing Synchronous Circuitry”, *Journal of VLSI and Computer Systems*, pp 41-67, 1983.

[6] C. Leiserson, and J. Saxe, “Retiming Synchronous Circuitry”, *Algorithmica*, 6(1):5-35, 1991.

[7] Larry McMurchie and Carl Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM Third International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995.

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

[9] A. Sharma, “Development of a Place and Route Tool for the RaPiD Architecture”, *Master’s Project, University of Washington*, December 2001.

[10] Amit Singh, Arindam Mukherjee, Malgorzata Marek-Sadowska, “Interconnect Pipelining in a Throughput-Intensive FPGA Architecture”, *ACM/SIGDA Ninth International Symposium on Field-Programmable Gate Arrays*, pp 153-160, 2001.

[11] Deshanand P. Singh, Stephen D. Brown, “The Case for Registered Routing Switches in Field Programmable Gate Arrays”, *ACM/SIGDA Ninth International Symposium on Field-Programmable Gate Arrays*, pp 161-169, 2001.

[12] Deshanand P. Singh, Stephen D. Brown, “Integrated Retiming and Placement for Field Programmable Gate Arrays”, *Tenth ACM International Symposium on Field-Programmable Gate Arrays*, pp 67-76, 2002.

[13] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek and Andre DeHon, “HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array”, *ACM Seventh International Symposium on Field-Programmable Gate Arrays*,pp , 1999.

APPENDIX A

We can show that the Two-terminal N-Delay Routing Problem (abbreviated here as 2TND) 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 edged (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 N, all solutions to the new problem will also have |V| edges, a cost of N+|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.

As stated in [2], TSP-TI is NP-Complete. We can reduce TSP-TI to 2TND by transforming all TSP-TI nodes to D-nodes, and converting the edge-weights of TSP-TI to R-nodes. 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 2TND. Let $Source_{TSP}$ 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 $M_{TSP}=Source_{TSP}$, 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 2TND with $N=|V_{TSP}|-1$, and S & K = $Source_{2TND}$, the node corresponding to $Source_{TSP}$.

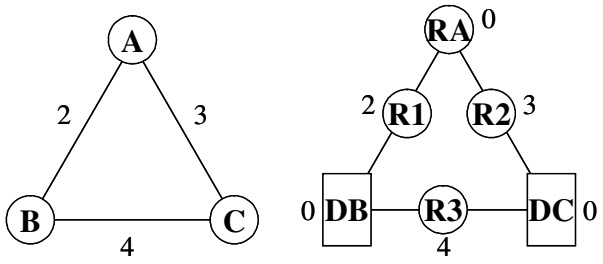

Figure A-1: Example TSP-TI (left) with edge weights, and the corresponding 2TND (right), with node weights. TSP-TI node A is chosen as the source & sink, and N=2.

We must now show that the solution to the 2TND problem gives us a solution to the TSP-TI problem. One concern is that the 2TND 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 2TND problems on the graphs created from TSP-TI problems, we will define *simplified* 2TND solutions. Specifically, walk the 2TND 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 2TND. 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 the type-D 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}->Rn_{2TND}->$Post_{2TND}$, 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 2TND 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 $Source_{TSP}$ at the beginning and end of the path. Finally, the cost of the path is no greater than the cost of the original 2TND solution.

The simplified 2TND 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 2TND, plus the $Source_{2TND}$ node, are a *tour* in TSP-TI. A tour is a simple cycle visiting all nodes in a graph exactly once. In our simplified 2TND solution all D-nodes are visited exactly once. By converting the path starting and ending at $Source_{2TND}$ into a cycle by fusing together the ends, you also visit $Source_{2TND}$ exactly once. The cost of the simplified 2TND 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 2TND solution with the same cost. Specifically, the tour in TSP-TI can be split at the $Source_{TSP}$ 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 2TND. 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 2TND solution will never violate the entry/exit rule of 2TND. Again, the cost of the TSP-TI and 2TND solutions are the same, since the edge weights of TSP-TI are identical to the node weights encountered in the 2TND solution.

As we have shown, all solutions of TSP-TI have a corresponding, equal cost solution in 2TND, and all simplified 2TND 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 2TND, then map the results of 2TND to a simplified 2TND result, and finally convert this into a solution to TSP-TI. Thus, since TSP-TI is NP-Complete, 2TND 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, 2TND is in NP. Since it is also NP-Hard, 2TND is therefore NP-Complete.

APPENDIX B

We assume that the FPGA architecture is represented as a simple, undirected graph that consists of R-nodes and D-nodes. The cost of each node in the graph is greater than or equal to one. An edge between two nodes in the graph merely represents a physical connection between the two nodes. Thus, all edges in the graph are unweighted.

The problem of finding an optimal 1-Delay path between two nodes in the graph is stated as:

Two-terminal 1-Delay Problem: *Let G=(V,E) be a simple, 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 one distinct D-node such that* $w(P_G(S,K))$ *is minimum, where*

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

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

In Section 3, we proposed the *2Combined-Phased-BFS* algorithm as an optimal solution to the two-terminal 1-Delay problem. Before explaining the proof for *2Combined-Phased-BFS*, we briefly summarize the algorithm. Fig. B-1 shows pseudo-code for *2Combined-Phased-BFS*. 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 marked ***finally_explored*** if *x0 == 1*. The sub-routine *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 K) the neighbors of *lnode* are added to *PQ* at phase 1.

```
2Combined-Phased-BFS(S,K){
    Init PQ to S at phase 0;
    LOOP{
        Remove lowest cost node lnode from PQ;
        if(lnode == 0){
            1 Delay path between S and K does not exist;
            return 0;
        }
        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 1-Delay path from S to K;
            else{
                if(x1 == 2)
                    Mark lnode finally_explored at phase 1;
                AddNeighbors(PQ,lnode,path_cost,1);
            }
        }
    }
}

AddNeighbors(PQ,lnode,path_cost,p){
    Loop over each 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. B-1: Pseudo-code for *2Combined-Phased-BFS*

We now prove that *2Combined-Phased*-BFS is in fact optimal. We present a proof by contradiction in which we show that *2Combined-Phased-BFS* will always find an optimal 1-Delay path between S and K, if one exists. Before we begin the proof, there is some terminology we will introduce to simplify things. The algorithm presented in this paper explores multiple paths through the graph via a modification to Dijkstra's algorithm. We state that our algorithm explored 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. Path A is "more explored" than path B if the cost of the identical path on A from source to A's last explored point is greater than the cost of the identical path on B from source to B's last explored point.

For purposes of this proof we will 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. B-2 shows the *most explored* optimal 1-Delay path between S and K. In other words, the path shown in the figure is the best 1-Delay path between S and K, with 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-Delay path between S and K goes through multiple D-nodes, then the D-node nearest S is used to pick up delay.

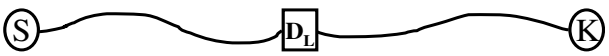

Fig. B-2: The initial assumption is that the most explored lowest cost 1-Delay route between S and K goes through D-node D_L

Although it appears that the paths S-D_L and D_L-K in Fig. B-2 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 the diagrams of this section, we use the convention of showing paths without overlaps (Fig. B-3), even though they may actually overlap (Fig. B-4). Our proof does not rely on the extent of intersection between hypothetical paths (which are always shown in gray) and the known best 1-Delay path.

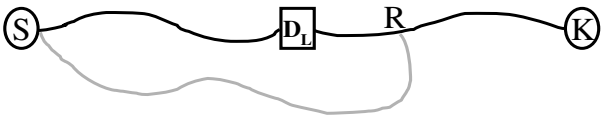

Fig. B-3: Representation of a path from S to node R shown in gray

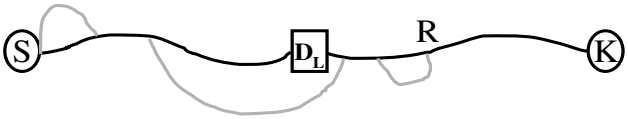

Fig. B-4: The path from S to R could actually intersect with the paths S-D_L and D_L-K

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

- **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.

We now show that none of the above-mentioned cases can occur, thus guaranteeing the optimality of *2Combined-Phased-BFS*. Each case is dealt with separately.

CASE 1: An R-node on the path S-D_L gets explored at phase 0 along a path other than S-D_L.

The cost of the gray path S-G-R (Fig. B-5) is 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. Thus, we have proved that **CASE 1** can never occur.

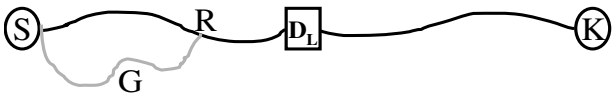

Fig. B-5: The case in which an R-node on the path S-D_L gets explored at phase 0 along some other path

CASE 2: D-node D_L gets explored at phase 0 along two paths other than S-D_L.

In Section 3 we demonstrated that if we allow a D-node to be visited only *once* at phase 0, *Combined-Phased-BFS* fails on the graph topology shown in Fig. 4. The reason for the failure can be seen in Fig. B-6. Assume that the cost of the path S-G-R-D_L shown in gray is less than the cost of path S-D_L along the known best path S-D_L-K. In this case, D-node D_L gets explored at phase 0 via the R-node that is used to exit D_L at phase 1 in the best path S-D_L-K. If we allow D-nodes to be explored at phase 0 only once, then the known best path S-D_L-K will not be found.

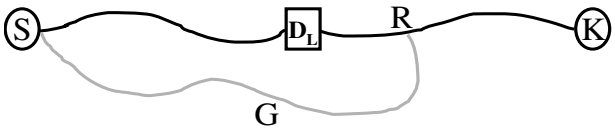

Fig. B-6: The cost of path S-G-R-D_L is less than the cost of path S-D_L along the known best path S-D_L-K

While D_L can get explored at phase 0 by *one* path other than S-D_L, we will now show that it is not possible to explore D_L at phase 0 along *two* paths other than S-D_L.

The node D_L could get explored at phase 0 twice in the following possible ways:

- Fig. B-7: 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 be better than the known best path S-D_L-K, thus contradicting our initial assumption.

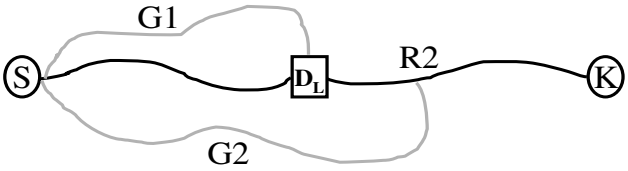

Fig. B-7: D_L gets explored at phase 0 along paths S-G1-D_L and S-G2-R2-D_L

- Fig. B-8: The cost of each of the paths S-G1-D_L and S-G2-D_L is less than or equal to the cost of path S-D_L. If D-node D_L gets explored at phase 0 along these two paths, both S-G1-D_L-K and S-G2-D_L-K would be better than the known best path S-D_L-K, which contradicts our initial assumption.

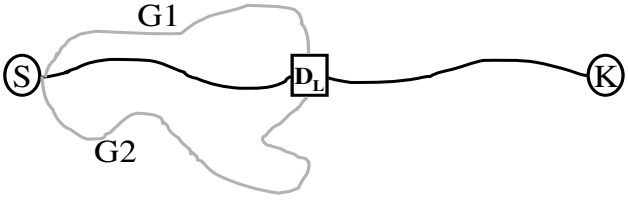

Fig. B-8: The cost of each of the paths S-G1-D_L and S-G2-D_L is less than or equal to the cost of path S-D_L

Finally, note that D_L can never get explored at phase 0 along both gray paths shown in Fig. B-9, regardless of the cost of these paths. This is because R-nodes can be explored only once at phase 0, which means that D_L can get explored at phase 0 by only *one* of the paths S-G1-R1-D_L or S-G2-R2-D_L. Therefore, it is not possible to explore D_L at phase 0 two times via the node that is used to exit D_L along the best path S-D_L-K.

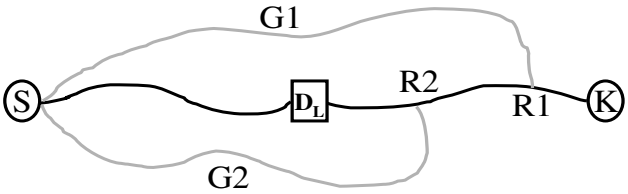

Fig. B-9: D_L cannot get explored at phase 0 along both S-G1-R1-D_L and S-G2-R2-D_L.

We have thus proved that it is not possible to explore D_L at phase 0 along *two* paths other than S-D_L. Thus, **CASE 2** can never occur.

CASE 3: A node on the path D_L-K gets explored at phase 1 along two paths other than D_L-K.

We will first enumerate the different cases in which a node X on the path D_L-K can possibly get explored at phase 1 along a path other than the known best path, before the known best path reaches that node:

- Fig. B-10: The D-node at which the path picks up delay does not lie on the known best path S-D_L-K. The figure shows the different 1-Delay paths on which node X can possibly get explored at phase 1:
  - Node X gets explored at phase 1 along the path S-D-G1-R1-D_L-X. This means that the path S-D-G1-R1-D_L-X-K is better than the known best path, thus contradicting our initial assumption.
  - Node X gets explored at phase 1 along the path S-D-G2-D_L-X. This means that the path S-D-G2-D_L-X-K is better than the known best path, which contradicts the initial assumption.
  - Node X gets explored at phase 1 along the path S-D-G3-X. This means that the path S-D-G3-X-K is better than the known best path, which is a contradiction of our initial assumption.

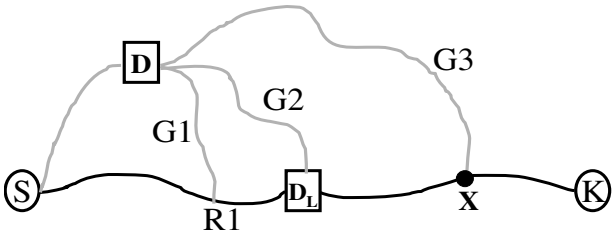

Fig. B-10: Node X could get explored at phase 1 along any of the three paths shown in gray. D-node D does not lie on the known best path S-D_L-K.

- Fig. B-11: The D-node at which the path picks up delay lies on the phase 0 segment of the known best path. The cost of path S-G1-D-D_L-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-D-D_L-X-K would be better than the known best path, thus contradicting our initial assumption.

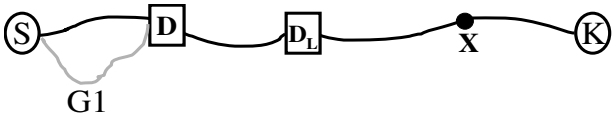

Fig. B-11: Node X gets explored at phase 1 along the path S-G1-D-D_L-X.

- Fig. B-12: The cost of path S-G1-D_L-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-G1-D_L-X-K is better than the known best path, which is a contradiction of our initial assumption.

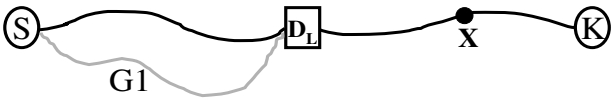

Fig. B-12: Node X gets explored at phase 1 along the path S-G1-D_L-X.

- Fig. B-13: The D-node at which the path picks up delay lies on the phase 1 segment of the known best path. Node X is closer to the sink K than D-node D. 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.

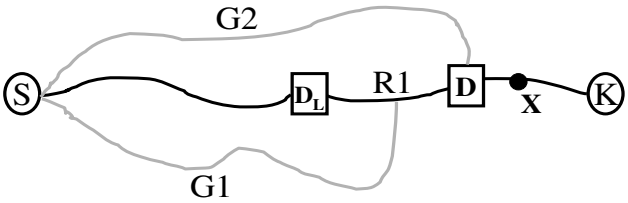

Fig. B-13: Node X can get explored at phase 1 along either S-G2-D-X or S-G1-R1-D-X.

- Fig. B-14: The D-node at which the path picks up delay lies on the phase 1 segment of the known best path. D-node is closer to sink K than node X. Again, there are two possibilities here:
    - 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. In this case, the path S-G2-D-K would be better than the known best path, thus contradicting our initial assumption.
    - The cost of the 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, node X gets explored at phase 1 along the path S-G1-R1-D-X. Note that this is the only case in which a node on the phase 1 segment of the known best path can get explored at phase 1 along a path other than the known best path. If we were to allow the nodes in our graph to be explored at most once at phase 1, this case represents a true failure case.

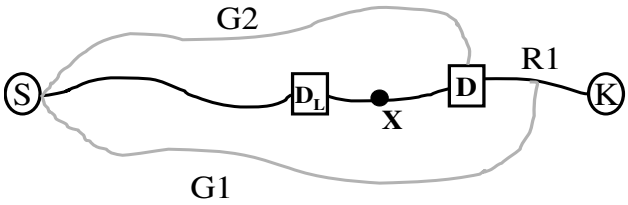

Fig. B-14: Node X can get explored at phase 1 along either S-G2-D-X or S-G1-R1-D-X.

Thus, we have proved that the only case in which a node on the phase 1 segment (path D_L-K in Fig. B-2) can get explored at phase 1 along a path other than the known best path is the path S-G1-R1-D-X shown in Fig. B-14. We now prove that it is not possible to have *two* such paths:

- Fig. B-15: The cost of each of the paths S-G1-R1-D-X and S-G2-R2-D'-R1-D-X is less than or equal to the cost of the path to X along the known best path. This means that the cost of the path to node R1 is less than or equal to the cost of the path to R1 along the known best path, which in turn implies that the path S-G1-R1-D'-R2-K is better than the known best path. This is a contradiction of our initial assumption.

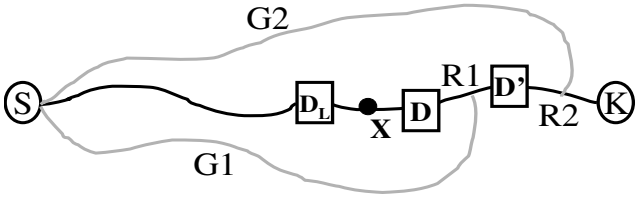

Fig. B-15: Node X can get explored at phase 1 along either S-G1-R1-D-X or S-G2-R2-D'-R1-D-X.

Hence, we have proved that a node on the segment D_L-K (Fig. B-2) of the known best path cannot get explored at phase 1 along *two* paths other than the known best path from S to K. Thus, **CASE 3** can never occur.

Furthermore, since we have proved that **CASE1, CASE2** and **CASE3** can never occur, the algorithm *2Combined-Phased-BFS* is optimal.