Enhancing Routing Heuristics on Pipelined-FPGAs

Ken Eguro and Scott Hauck

Abstract— While previous research has shown that FPGAs can effectively implement many types of computations, their flexibility inherently limits maximum clock rates. To minimize the effect of this on throughput, circuit designers often deeply pipeline designs whenever possible. Several research groups have attempted to support this type of heavily pipelined circuit by developing new architectures that include registered switchpoints within the interconnect. Unfortunately, this pipelined communication network presents a new and difficult problem for detailed routing tools. In this paper we discuss the details of this problem and present a new timing-driven pipeline-aware router that produces 40% better critical path delay than previous efforts.

***Index Terms*— Algorithms, design automation, field programmable gate arrays, reconfigurable architectures**

I. Introduction

Although it has been long known that FPGAs bridge the gap between flexible, but relatively slow software running on general-purpose processors and extremely fast, but costly ASICs, the programmable nature of FPGAs introduces significant inefficiencies that limit the clock frequency of mapped circuits. That said, many research efforts have shown that the flexibility of reconfigurable devices can outweigh such speed penalties. However, designers would often like to compensate for the naturally lower clock frequency of FPGAs by heavily pipelining, retiming, and C-slowing their computations whenever possible. Unfortunately, these techniques often produce a large number of registers that conventional FPGA architectures cannot adequately deal with.

Multiple research groups have attempted to provide better support for heavily pipelined circuits and increase the achievable clock frequency by developing specialized pipelined FPGA architectures. A characteristic feature of these systems is additional registering resources embedded within the interconnect network itself. HSRA [9], RaPiD [1], and an architecture by Singh and Brown [8] are good examples of this approach. However, although these devices have been shown to improved throughput and efficiency for many types of applications, the introduction of registers into the interconnect network can fundamentally change the nature of the placement and routing problem.

Register assignment is straightforward for customary netlists on conventional FPGAs, since pipelining resources are flip-flops embedded within logic blocks, generally with flexible input multiplexing. This is not necessarily true for pipelined architectures. In a pipelined-FPGA, assigning registers at packing or placement time may not result in reasonable quality. This is because in a pipelined architecture many of the available registers are inside interconnect elements such as switchboxes. To allow for an efficient layout, these interconnect registers will generally have very little or no flexibility in their connectivity – completely unlike those found in logic blocks. This means that assigning a signal's registers during placement could be tantamount to determining the majority of a signal's detailed routing during placement. At best, this will dramatically affect the routability of deeply pipelined netlists since so many of the signals will essentially be locked to specific routing tracks and outside the control of normal detailed routing techniques. At worst, as is likely on a track-graph FPGA, unconstrained register assignment will likely lead to unroutable circuits.

While this would suggest that we should assign registers during the routing process, this inherently changes the routing problem itself. It is no longer simply a matter of finding a path between a source and sink, we now need to find a path between a source and sink that goes through exactly N registers. Formally introduced in [7], this problem is known as the *N-Delay Routing* problem. Although the authors of [6] and [3] presented two different heuristics to discover pipelining registers during routing, both of these approaches perform purely congestion-driven routing. Since heavily pipelining the netlists and augmenting the architectures with additional registering resources were originally motivated by timing concerns, not considering timing during routing will likely lead to disappointing results.

Although timing-driven heuristics have been well studied for conventional FPGA routing [4], there are several details of the N-Delay Routing problem that prevent us from simply using classical timing-driven cost formulations inside existing pipeline-aware routing algorithms. In this paper we will discuss some these limitations and present a new timing-driven pipeline-aware router.

II. The N-Delay Routing Problem and Pipeline-Aware Routing Heuristics

PathFinder [4] and its predecessors demonstrated that the conventional routing problem for multi-terminal, multi-net circuits is very difficult. However, it breaks down into much simpler sub-problems. For example, if we ignore congestion temporarily, we can use Dijkstra's shortest-path algorithm to easily find routes for two-terminal nets. PathFinder capitalizes heavily on this and augments Dijkstra's algorithm to handle multi-terminal nets and resolve congestion.

However, since the N-Delay Routing problem adds the additional constraint of a required pipelining latency, Dijkstra's algorithm cannot be used. First, the shortest path from source to the sink may not meet the specified latency requirement. More importantly, the *shortest* path to any given node along the way may not be the *best* path since it may not form the prefix of any legal route. For example, in Fig. 1 we would like to find a path between the source *S* and the sink *K* that goes through exactly one pipelining register.

Manuscript received March 14, 2006; revised on June 20, 2006

If we assume a unit cost model, we can see that Dijkstra's algorithm fails to find a valid one-latency path. This is because node *f* is explored first by the zero-latency search from (*S*, *d*, *e*). Since Dijkstra's algorithm marks all nodes when they are visited, the initially more expensive (*S*, *a*, *b*, *c*) route is blocked from continuing on to the sink. As discussed in [5], finding a legal N-Delay route is NP-Complete.

As far as we are aware, only two previous research efforts have been made to address the N-Delay Routing problem: PipeRoute [6] and QuickRoute [3]. Although both utilize PathFinder's iterative *Negotiated Congestion* methodology in an outer loop to gradually discourage sharing, they have each replaced PathFinder's inner Dijkstra's search with different heuristics to discover pipelined paths.

A. PipeRoute

PipeRoute forms multiple latency paths by iteratively combining single-register routes. In [6], the authors show a technique to find one-latency routes in polynomial time and they use this methodology to slowly build N-Delay paths. As seen in Fig. 2, to find a two-latency path from the source *S* to the sink *K*, they first attempt to find a one-latency path. If this initial single-register route elects to use register *f*, the next step is to attempt to replace either the link from *S* to *f* or *f* to *K* with its own lowest cost one-latency route.

Unfortunately, as discussed in [2], PipeRoute's fundamental one-latency router encounters problems on some architectures. PipeRoute's methodology hinges on the ability for a Dijkstra-like search to visit every node multiple times: once at latency zero (pre-register), and twice at latency one (post-register). Shown in the top and middle illustrations of Fig. 3, this allows a phase zero and phase one search to pass each other in order for one to find a complete path around a ring. Unfortunately, this also allows a path to discover a route that intersects itself. In the bottom illustration of Fig. 3 we see that the shortest one-latency paths are routes that visit a register, then double back onto themselves. Obviously, these are not valid physical routes, but as described in [2], we believe that PipeRoute will continually find this type of self-congesting path on most FPGA architectures, despite PathFinder's best efforts.

B. QuickRoute

Instead of gradually assembling higher latency routes from multiple smaller latency segments, QuickRoute attempts to find full N-latency paths directly. Although performed for latencies higher than only zero and one, it searches in a manner similar to PipeRoute in that a Dijkstra-like search is capable of visiting a node multiple times at phases 1 to N. A wave is allowed to explore a given node if the node has been visited by fewer than *k* other waves at the same latency. For example, in the top illustration of Fig. 4, if we assume *k*=2, the paths (*S*, *a*, *b*) and (*S*, *e*, *b*) would both be considered.

However, unlike PipeRoute, QuickRoute does not allow paths to intersect themselves. To accomplish this, it records the upstream nodes for every exploration and does not allow a wave to revisit a node already used by itself earlier in the search. On the bottom of Fig. 4, a path that goes through *b* will not consider it again for subsequent exploration.

Unfortunately, QuickRoute cannot guarantee a solution in all cases. We can see that in Fig. 5, if we assume that *k*=1 and we would like to go from *S* to *K* with a delay of two, we will fail to find a solution. This is because node *b* is initially used by a doomed route that, in turn, prevents a valid route from exploring *d*. No matter how large a *k* is used, a graph can be constructed that will self-block by adding additional

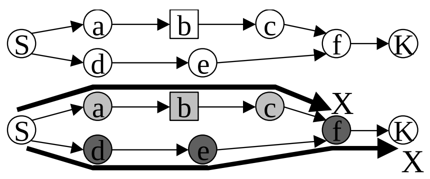


Fig. 1. Illustration of the N-Delay Routing Problem. Circles denote routing nodes, squares denote registers.

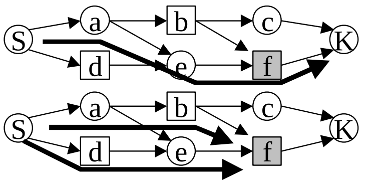


Fig. 2. Illustration of the PipeRoute Algorithm. Finding a one-latency route (top), expanding to a two-latency route (bottom).

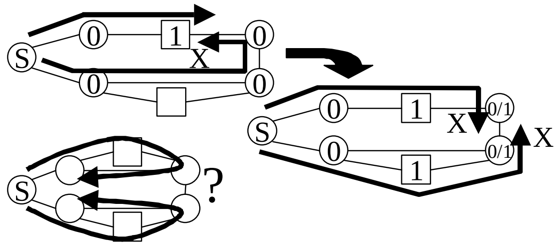


Fig. 3. PipeRoute Search Techniques and Limitations. Both phase zero (top) and phase one (middle) searches deadlock unless we allow multiple visitation. Notations indicate latency phase of visitation. Assume a search enters a register as a phase zero exploration and exits as a phase one. However, this causes a self-intersection problem (bottom).

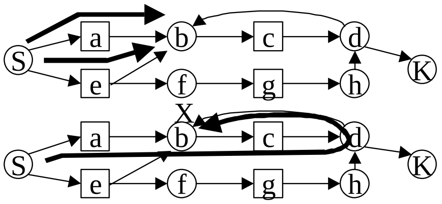


Fig. 4. Illustration of QuickRoute Algorithm. Multiple visitation (top), but no self-intersection (bottom).

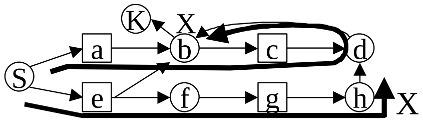


Fig. 5. Illustration of QuickRoute Self-Blocking.

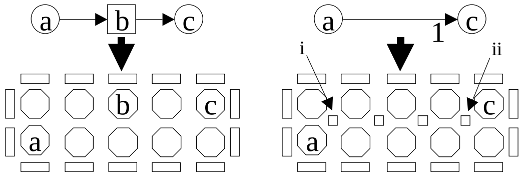
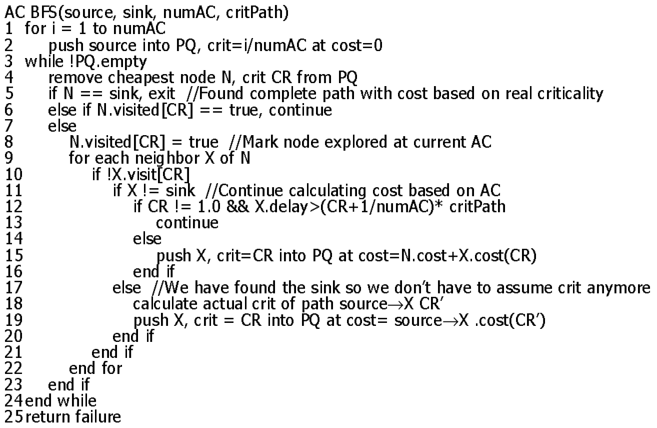


Fig. 6. Register Assignment and Criticality. Placement of registers in a conventional FPGA (left) and discovering registers during routing in a pipelined-interconnect FPGA (right).



```
AC BFS(source, sink, numAC, critPath)
for i = 1 to numAC
    push source into PQ, crit=i/numAC at cost=0
while !PQ.empty
    remove cheapest node N, crit CR from PQ
    if N == sink, exit  //Found complete path with cost based on real criticality
    else if N.visited[CR] == true, continue
    else
        N.visited[CR] = true  //Mark node explored at current AC
        for each neighbor X of N
            if !X.visit[CR]
                if X != sink  //Continue calculating cost based on AC
                    if CR != 1.0 && X.delay>(CR+1/numAC)* critPath
                        continue
                    else
                        push X, crit=CR into PQ at cost=N.cost+X.cost(CR)
                    end if
                else  //We have found the sink so we don’t have to assume crit anymore
                    calculate actual crit of path source→X CR’
                    push X, crit = CR into PQ at cost= source→X .cost(CR’)
                end if
            end if
        end for
    end if
end while
return failure
```

Fig. 7. Assumed Criticality Breadth-First Search.

registered paths between *b* and *d*. This said, as discussed in [2], we believe that QuickRoute holds an advantage over PipeRoute. Not only does QuickRoute defend itself from self-intersection, it has the flexibility to improve its routing ability by increasing *k*.

III. Determining Link Criticality

Although PipeRoute and QuickRoute provided the first solutions to the pipelined-routing problem, neither considered delay. This might seem surprising considering the maturity of conventional timing-driven routing techniques, but fundamental differences between the classical and the pipelined routing problems prevent us from directly leveraging PathFinder’s timing-driven cost formulation.

In [4], PathFinder defines (1) – the cost of a node (C_n) is not only related to its delay (d_n) and congestion (c_n), but also to the criticality of the source/sink pair (A_{ij}) as determined in the last routing iteration.

$$C_n = A_{ij} d_n + (1 - A_{ij}) c_n \quad [4] \tag{1}$$

Since A_{ij} falls between zero and one, a timing-critical net (A_{ij}=1) only considers the delay of a node without considering its congestion cost. In this way, it will naturally seek the fastest possible route. However, as A_{ij} approaches zero, congestion will pay a larger and larger role in determining which path is taken.

This formulation has proven itself very effective in conventional FPGAs. However, it leads to gross system instability when applied to pipelined architectures. As shown on the right of Fig. 6, in a pipelined FPGA we must find registers as part of the routing process itself. Because of this, the criticality of the pre and post-register links will heavily depend upon the registering location that is chosen. Obviously, the relative criticality of these links will change completely if we choose to register at *i* versus *ii*. The classical PathFinder methodology now shows an unusual timing oscillation as opposite sides of the register vie for dominance.

If the first iteration chooses to register at *i* we can guarantee that the second iteration will choose to register at *ii*, despite better options. This occurs because the pre-register link will have a very low criticality, making delay on this segment during the next routing iteration very inexpensive. Conversely, the post-register link will have a very high criticality making delay very costly. Ignoring congestion temporarily, the post-register link will want to become as short as possible and low delay cost on the pre-register link facilitates this.

Essentially, this type of behavior occurs because we utilize old and dramatically incorrect net criticality information to determine future routes. The criticality of a link used in one routing iteration has no bearing in the next if we select a different register. Looking at the problem from the standpoint of conventional routing, we should not be surprised at the instability of the system – it is as if we can change the placement of all the registers between every routing iteration.

Clearly, if we would like to obtain high quality results, we cannot use previous criticality information to guide future exploration. However, we still need some mechanism to balance congestion with delay. We suggest that an exploration decides the proper criticality for a route at the only point that the decision can actually be correctly made – when it arrives at a sink.

In this formulation, we start *AC* independent searches from the source, each assuming the net has a different criticality ranging from 1/*AC* to 1.0. In this manner, we will have multiple simultaneous searches that each emphasize delay versus congestion in a slightly different way. In theory, the first exploration to reach the sink will be the least expensive and, thus, represent approximately the proper balance of congestion versus delay. However, the real criticality of a path must be incorporated into the cost calculation somehow to ensure accuracy between the perceived and actual cost of a route. Otherwise, high congestion can be easily hidden along non-critical paths by simply assuming that the net is critical. Similarly, high delay can be hidden along critical paths by simply assuming that the net is not critical.

Instead, as shown in Fig. 7, the assumed criticality should be used to calculate the cost of route up to, but not including the sink. At this point, we no longer need the assumed criticality since the real delay of the route can be determined and the true cost of the path can be calculated. This will

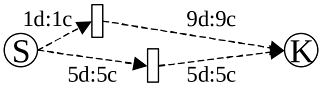


Fig. 8. Congestion Versus Timing Cost Dependency of PathFinder Cost Formulation. Notation is (delay:congestion) and we define *c* and *d* to be the average base delay and congestion cost for the architecture. We assume a circuit-wide critical path of 10d.

ensure that a sink is only push into the search queue with the true cost of the path.

This would prevent the potential scenario in which low assumed criticality searches rush ahead along uncongested, but slow links and form an unnecessarily high criticality path. Just as these searches are about to reach the sink, they will calculate the real criticality of the paths found. The cost of these searches will then rise dramatically to reflect their newly revealed high delay and high criticality. This will allow higher assumed criticality searches, that might find faster, slightly more congested paths, to catch up and have the opportunity to form a more appropriate mid-criticality link.

Furthermore, to improve the runtime of this technique, we can prune non-productive searches. With the exception of the highest criticality exploration, we can prune a search when the current path delay would make the exploration's criticality larger than the next higher assumed criticality search. For example, for *AC*=5 we will launch five explorations with criticalities (0.2, 0.4, 0.6, 0.8, 1.0). If the current critical path is 10, paths with a delay of 4 or more do not need to be explored by the 0.2 assumed criticality wave. Those paths would be better serviced by the 0.4 assumed criticality exploration.

IV. Revised Cost Formulation

Unfortunately, although the assumed criticality methodology addresses routing errors due to inaccurate link criticality information, the classical timing-driven PathFinder cost formulation still creates problems when applied to pipelined routing. These issues are related to how (1) intertwines an architecture's base delay and congestion costs.

Looking at Fig. 8, we see two potential one-latency paths from *S* to *K*. Both paths have the same total congestion and delay, but the top path is a relatively poor choice because the post-register link is both very close to critical and highly congested. If we define the overall cost of a pipelined route to simply be the summation of the cost of each individual link from (1), we can see that the cost for these routes become (2) and (3).

$$0.1(d)+0.9(c)+0.9(9d)+0.1(c)=8.2d+c \quad (2)$$

$$0.5(5d)+0.5(5c)+0.5(5d)+0.5(5c)=5d+5c \quad (3)$$

From these equations, we can see that the selection of balanced versus unbalanced paths depends entirely upon the relative values of *c* and *d*, an architecture's average base delay and congestion cost. In our example, the more balanced path is only selected if *c* < 0.8*d*. Unfortunately, even if the base costs of the architecture are initially scaled correctly, the natural congestion cost escalation of PathFinder will cause later iterations to tend toward worse selections. These unbalanced paths will actually work contrary to PathFinder's own attempts at congestion resolution. As we enter the later stages of routing, the average congestion cost will rise to resolve sharing. However, based upon our observation, we will actually tend towards more heavily congested options.

This problem occurs because the delay and congestion contributions to the overall path cost are linked. While the A_{ij} versus (1-A_{ij}) terms guarantee that paths can trade delay for congestion and vice-versa, this intertwines the two components making their relative values very sensitive. To address this issue, we make a subtle change that removes this vulnerability. If we divide both sides of (1) through by (1-A_{ij}), we obtain (4).

$$\frac{C_n}{1-A_{ij}}=\frac{A_{ij}d_n}{1-A_{ij}}+\frac{(1-A_{ij})c_n}{1-A_{ij}}=\frac{A_{ij}}{1-A_{ij}}d_n+c_n \quad (4)$$

This does not change results for unpipelined signals, since all paths are simply divided by the same (1-A_{ij}) factor. Revisiting our pipelined example from Fig. 8 we get (5) and (6).

$$0.11(d)+c+9(9d)+9c=81.11d+10c \quad (5)$$

$$1(5d)+5c+1(5d)+5c=10d+10c \quad (6)$$

Since both the congestion and delay costs are necessarily positive numbers, we can see that more balanced paths are now always selected over unbalanced paths. With this change, the cost of an L-latency path becomes (7), the summation of the congestion encountered within each link, plus the relative delay of each segment adjusted by its individual criticality.

$$C=\sum_{i=0}^{L}\text{timingCost}_i+\sum_{i=0}^{L}\text{congestionCost}_i \quad (7)$$

V. Results

In [2], the assumed criticality methodology and revised timing-driven cost formulation are combined with QuickRoute-based searches to make the Armada timing-driven pipelined routing algorithm. To determine the effectiveness of this new algorithm, we compare ourselves to both PipeRoute and QuickRoute on the architecture that first inspired the pipelined routing problem, RaPiD. Our evaluation encompasses nine RaPiD netlists that represent a wide range of pipeline register requirements. These were

Table I
Normalized Results for Original Length-16 RaPiD Architecture

	Algorithm	Tracks	Timing	Runtime	Sec/It
Best Track	*PipeRoute-TD*	1.09	1.56	**0.09 : 90 s**	**0.09**
	QuickRoute	1.04	1.64	0.10 : 133 s	0.18
	Armada	**1.00**	**1.00**	1.00 : 1721 s	1.00
	Armada-PFind	1.03	1.18	7.65 : 7982 s	4.51
Match Track	*QuickRoute*	1.05	1.73	**0.11 : 138 s**	**0.18**
	Armada	1.05	**0.99**	1.05 : 1826 s	1.07
	Armada-PFind	1.05	1.20	7.45 : 7705 s	4.72
Normalized Results for Length-8 Long Tracks					
Best Track	*PipeRoute-TD*	1.00	1.66	0.09 : 113 s	**0.06**
	QuickRoute	**0.96**	1.65	**0.10 : 66 s**	0.12
	Armada	1.00	**1.00**	1.00 : 1357 s	1.00
	Armada-PFind	1.02	1.30	3.11 : 3075 s	3.83
Match Track	*QuickRoute*	1.03	1.71	**0.08 : 45 s**	**0.14**
	Armada	1.03	**1.00**	0.88 : 841 s	1.07
	Armada-PFind	1.03	1.31	3.12 : 3075 s	3.84
Normalized Results for Length-4 Long Tracks					
Best Track	*PipeRoute-TD*	1.01	1.59	**0.02 : 53 s**	**0.03**
	QuickRoute	1.02	1.54	0.11 : 76 s	0.11
	Armada	**1.00**	**1.00**	1.00 : 2637 s	1.00
	Armada-PFind	1.05	1.21	2.75 : 2976 s	1.81
Match Track	*QuickRoute*	1.05	1.55	**0.10 : 41 s**	**0.10**
	Armada	1.05	**0.99**	0.84 : 1593 s	1.22
	Armada-PFind	1.05	1.21	2.75 : 2976 s	1.81

Average normalized track counts, normalized critical path delay, normalized and raw algorithm runtime and normalized seconds required per routing iteration. All normalized results are normalized to *Best Track* Armada values.

mapped to three different architectures: the original RaPiD architecture and two others that increase the density of pipelined switchpoints.

To gather our results, we first ran all nine netlists through the placer built into PipeRoute [6]. This provided a fixed, pipelining-aware placement as a starting point for all three routing algorithms. Six independent PipeRoute placement and routing runs were performed and the best results were gathered. The PipeRoute router actually used was an improved version from [5] that augmented the original PipeRoute algorithm with a rudimentary timing-driven formulation similar to that of PathFinder. These placements were then also routed using the original congestion-driven QuickRoute algorithm, the Armada algorithm, and the Armada algorithm substituting in the original PathFinder cost formulation.

In Table I, *Best Track* results are the average normalized track requirements, circuit timing and router runtime when each tool searches separately for the minimum routable architecture for each of the nine netlists. *Match Tracks* results are obtained when each tool is given the maximum number of tracks required by any of the QuickRoute-derivative tools for a given netlist. Although a precise relationship cannot be made due to the wide range of benchmark complexity, we also include un-normalized average router runtime to give a general sense of algorithm effort. We also include normalized seconds per iteration results to reflect the comparative difficulty between a standard QuickRoute search and the Armada variants. This is intended to roughly reflect the relative efficiency of each search technique. All results were gathered on 3.2GHz Intel Xeon machines with 2GB of RAM.

As seen in Table I, the Armada algorithm finds vastly superior timing results with essentially identical routability. Although it requires somewhere between 5x and 10x more runtime relative to QuickRoute, previous approaches generally have 60% worse critical path timing. We can also see that our new timing-driven cost formulation functions as intended when we substitute PathFinder's back into the Armada algorithm (Armada-PFind). We can see that this change alone is responsible for at least a 15% improvement in critical path delay. Surprisingly, replacing the cost function also dramatically changes the single iteration search time. We believe that the original PathFinder cost function somewhat flattens the cost of the various possible paths leading to a less directed search than the Armada cost function.

VI. Conclusions and Future Work

We have shown in this paper that the N-Delay Routing problem presents inherently different challenges from the classical one. Simultaneous register assignment significantly complicates the normal issues of timing-aware congestion resolution. To address these issues we developed a new methodology to determine path criticality and a new timing-driven cost formulation. Leveraging aspects from a host of previous techniques, the Armada timing-driven pipeline-aware routing algorithm was created. On three RaPiD architectures this algorithm was shown to provide roughly 40% better average timing results without affecting routability. While more computationally intensive than previous algorithms, Armada remains competitive, especially given the dramatic timing benefit.

References

[1] C. Ebeling, D. Cronquist and P. Franklin. "RaPiD -Reconfigurable Pipelined Datapath". *6th International Workshop on Field-Programmable Logic and Applications*, 1996: 126-35.

[2] K. Eguro and S. Hauck. "Armada: Timing-Driven Pipeline-Aware Routing for FPGAs". *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2006: 169-78.

[3] S. Li and C. Ebeling. "QuickRoute: A Fast Routing Algorithm for Pipelined Architectures". *IEEE International Conference on Field-Programmable Technology*, 2004: 73- 80.

[4] L. McMurchie and C. Ebeling. "PathFinder: A negotiation-based performance-driven router for FPGAs". *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 1995: 473-82.

[5] A. Sharma, *Place and Route Techniques for FPGA Architecture Advancement*, Ph.D. Thesis, University of Washington, Dept. of EE, 2005.

[6] A. Sharma, C. Ebeling and S. Hauck. "PipeRoute: A Pipelining-Aware Router for FPGAs". *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2003: 68-77.

[7] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", *University of Washington, Dept. of EE Technical Report UWEETR-2002-0018*, 2002.

[8] D. Singh and S. Brown. "The Case for Registered Routing Switches in Field Programmable Gate Arrays". *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2001: 161-9.

[9] 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 Symposium on Field Programmable Gate Arrays*, 1999: 125-34.