Armada: Timing-Driven Pipeline-Aware Routing for FPGAs

Ken Eguro and Scott Hauck
Electrical Engineering Department
University of Washington
Seattle, WA
{eguro, hauck} @ee.washington.edu

ABSTRACT

While previous research has shown that FPGAs can efficiently implement many types of computations, their flexibility inherently limits their clock rate. Several research groups have attempted to address this by developing new architectures that include registered switchpoints within their interconnect. Unfortunately, this pipelined communication presents a new and difficult problem for detailed routing tools. Known as the N-Delay Routing Problem, it has been proven to be NP-Complete. Although there have been two heuristics developed to address this issue, both have certain limitations and neither approach considers timing during the routing process. While timing-driven conventional routing is largely considered to be a solved problem, there are several issues inherent to the N-Delay Routing problem make addressing timing particularly difficult. In this paper we discuss the nature of these problems and present a new timing-driven pipeline-aware router that produces as much as 60% better critical path delay than previous efforts.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids – *Placement and routing.*

General Terms

Algorithms, Performance, Design.

Keywords

Pipeline Routing, Pipeline FPGA, Reconfigurable Computing.

1. Introduction

Although it has been long known that FPGAs effectively 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. While this limitation might not affect designs typical of FPGAs' traditional role, such as control or glue logic, this commonly dissuades some designers from using FPGAs to perform computation-heavy datapath calculations. However, research into reconfigurable computing platforms has shown that in many cases the advantages of flexibility, ease of programming, and ability to reuse silicon can overcome the speed penalties generally incurred when using reconfigurable devices.

One constraint that hampers the further progress of reconfigurable computing systems is that designers would like to compensate for the naturally lower clock frequency of FPGAs by heavily pipelining, retiming, and C-slowing their computations when possible. Unfortunately, these techniques often produce a large number of registers that conventional FPGA architectures and development tools cannot adequately deal with. Commercial reconfigurable devices are designed with average-case circuits in mind so they do not include any pipelining resources beyond those that might be found in unoccupied logic block locations. This generally restricts retiming and pipelining to fairly minor changes, such as the techniques seen in [8].

Multiple research groups have attempted to address this problem with specialized pipelined-FPGA architectures. Since interconnect delay typically dominates most mapped circuits, one common solution has been to incorporate additional registering resources within the interconnect network itself. HSRA [10], RaPiD [2], and an architecture by Singh and Brown [9] are good examples of this approach. Unfortunately, the introduction of registers into the interconnect network can present some problems for existing CAD tools. While register assignment can be handled during the normal course of placement for registers in logic block locations, this might not result in reasonable quality for registers within the interconnect network. This is because these embedded registers will generally have very little or no flexibility in their connectivity. Thus, assigning registers during placement could be tantamount to performing detailed routing during placement. Although this may not present a problem if the number of pipelined signals is kept very small, this will dramatically affect the routability of deeply pipelined netlists.

While this would suggest that we should assign registers during the routing process, this fundamentally changes the problem of routing itself. No longer is it 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 proven to be NP-Complete, the authors of [6] and [3] presented two different heuristics to discovering pipelining registers during routing: *PipeRoute* and *QuickRoute*, respectively. Unfortunately, both of these approaches perform purely congestion-driven routing. Since heavily pipelining the netlists and augmenting the architectures with additional registering resources were both originally motivated by timing

FPGA'06, February 22-24, 2006, Monterey, California, USA.
Copyright 2006 ACM 1-59593-292-5/06/0002...$5.00.

concerns, not addressing timing during routing will likely lead to disappointing results.

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

2. N-Delay Routing Problem

While PathFinder [4] and its predecessors demonstrated that the conventional routing problem of congestion resolution for multi-net circuits is very difficult on most modern FPGA architectures, it breaks down into much simpler sub-problems. For example, if we ignore congestion altogether, we can use Dijkstra's shortest-path algorithm to quickly find routes between all sources and sinks. PathFinder borrows heavily from this and uses Dijkstra's algorithm with one modification – it updates node costs between routing iterations to gradually penalize overused resources.

Unfortunately, the N-Delay Routing problem adds the additional constraint of a required pipelining latency between source and sink. This precludes the use of Dijkstra's algorithm for two reasons. 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 even form the prefix of any legal route. For example, in Figure 1 we would like to find a path between the source *S* and the sink *K* that goes through exactly one pipelining register. If we assume a unit cost model, we can see that Dijkstra's algorithm fails to find a valid one-latency path. Obviously, (*S*, *d*, *e*, *f*, *K*) is a zero-latency path, so it does not fulfill the one register requirement. However, the reason that Dijkstra's does not find the valid path through node *b* 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, this prevents the initially more expensive (*S*, *a*, *b*, *c*) route from continuing on to the sink. As discussed in [4], finding legal N-Delay routes is NP-Complete. This problem becomes even more complicated when we consider building Steiner Trees for multi-terminal, multi-latency nets.

3. Pipelined FPGAs and Register Assignment

Given that the N-Delay Routing problem is so difficult, and workable solutions [3][6] have only been developed recently, how have previous research efforts dealt with this issue? Primarily, the problem has been avoided entirely by careful planning at the design level. For example, despite notable differences between the architectures, HSRA [10] and the pipelined-FPGA architecture proposed by Singh and Brown [9] both make specific architectural and toolflow decisions outside of the router to ensure legal pipelined paths. Unfortunately, these techniques ultimately limit the overall efficiency of the devices for heavily pipelined applications.

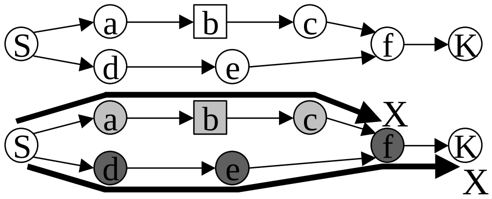

Figure 1. Illustration of the N-Delay Routing Problem. Circles denote routing nodes, squares denote registers.

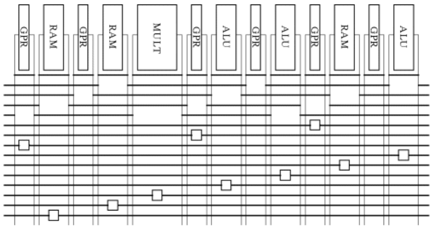

Figure 2. Illustration of a RaPiD Cell. Logic blocks populate the top of the array with short and long tracks below. Small squares represent optionally registered switchpoints between long tracks. Larger arrays are formed by abutting multiple RaPiD cells side-by-side.

HSRA is a hierarchical FPGA in which some fraction of the switchpoints are optionally registered. In addition, pipelining resources are provided at each logic block to balance the latency acquired in the interconnect – all LUT inputs have a large bank of optional flip-flops. By making the depth of these register banks equal to maximum number of pipelining stages along the longest path in the interconnect network, the HSRA designers avoid the N-Delay Routing problem. This is because if sufficient latency cannot be accumulated within the interconnect network itself because the path does not traverse enough registered switchpoints, the deficit can be satisfied by using the bank of registers at the destination pin. Unfortunately, this arrangement requires a huge number of registers on logic block inputs. For deeply pipelined netlists, as shown in [10], this incurs a hefty 2x to 5x area penalty.

Similarly, the system developed by Singh and Brown ensures that register assignment can be ignored during routing. However, their overall approach is slightly different. Their architecture is an island-style track-graph FPGA in which some fraction of the routing planes have optionally registered switchpoints creating pipelined and unpipelined track domains. In their toolflow, a designer first conventionally routes a circuit, then retimes the netlist after it is routed. However, to ensure sufficient pipelining resources along the already determined paths, the number of registers that can be pushed onto a given link during the retiming process must be constrained to the number of optional registers that currently exist along the route. Although this approach is a concise, closed-form solution, this technique greatly limits many of the advantages of retiming itself. Not only does this restrict the retiming of long paths to the available balancing resources that might be along associated short paths, we cannot take full advantage of many of the registers that the architecture already provides. Links that would like additional latency cannot acquire it even if there are unused interconnect registers neighboring the existing route.

To develop an efficient pipelined-FPGA architecture and to make best use of the provided registers likely requires a solution to the N-Delay Routing problem. In contrast to the previous architectures and their associated CAD tool suites, RaPiD [1] meets this problem head-on and was the inspiration of previous pipelined routing research. RaPiD is a coarse-grain, one-dimensional array with a word-width interconnect network. As seen in Figure 2, it contains a mixture of both short and long-distance tracks. Short tracks can be used for local, unpipelined communication. Long tracks can be concatenated for chip-wide routes and can acquire between zero and three retiming latencies at each switchpoint, also known as a *bus connector*.

In the existing toolflow, a high-level language compiler produces a retimed netlist that must be mapped to a device given specific link latency requirements. Although RaPiD architectures contain a wealth of pipelining locations, any specific bus connector can only communicate with the two wires immediate to its left and right. In this way, we can see that register assignment cannot be performed during placement. Deciding exactly which registers should be used also mostly determines the detailed routing for all pipelined signals. Unfortunately, deferring register assignment until routing also presents a problem since it is not obvious how to find a route between every source and sink that contains exactly the correct number of pipelining registers. We cannot utilize a conventional router because we have limited pipelining resources that determine the overall characteristics of each path. For example, logic blocks that are placed physically close to each other may not be able to be connected via the most direct route. If the connection between these blocks requires multiple pipelining delays, we need to take a more circuitous route to acquire sufficient registering.

4. Pipeline-Aware Routing Heuristics

As far as we are aware, only two 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* cost formulation in an outer loop to gradually discourage sharing, they have each modified PathFinder's inner loop in different manners to discover pipelined paths.

4.1 PipeRoute

PipeRoute forms multiple latency paths by iteratively combining single-register routes. Since the authors show that they can find optimal one-latency routes in polynomial time, they use this to their advantage. As seen in Figure 3, 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 *i*, the next step is to attempt to replace either the link from *S* to *i* or *i* to *K* with its own one-latency route. In our example, we would select the lowest cost alternative between the routes (*S*→*e*→*i*) and (*S*→*g*→*i*). Unfortunately, this is a greedy accumulation process. Clearly, if we required a three-latency route and we selected an interim two-latency path using registers *g* and *i*, we would be unable to find a valid route. This is because there is no way for the links (*S*→*g*), (*g*→*i*), or (*i*→*K*) to be replaced with its own single latency link. While this may present a problem, particularly on certain architectures, of far greater concern are the difficulties PipeRoute might encounter finding the initial single-latency paths.

Although the authors show a method to find optimal one-latency routes in polynomial time, this approach has some subtle yet serious limitations. They begin by showing that a normal breadth-first search is not sufficient considering the difference between pre-register and post-register routing. As seen in the top illustration in Figure 4, if *S* is both the source and sink, we will not find a valid one-latency path if we simply mark nodes visited or not visited. This is because neither search can pass the other to complete a path around the ring. Instead, we must also note the associated latency when a node is explored. That is, a post-register wave can expand to a given node even if it has already been explored by a pre-register wave. This is called a *Combined-Phased BFS*. However, the authors go on to show that even this is not entirely adequate. If we consider the bottom illustration in Figure 4, we can see that even if we allow nodes to be visited both at latency zero and latency one separately, we can enter a similar deadlock. In [7] the authors show these problems can be avoided and they can guarantee optimality if we allow nodes to be visited once at latency zero and twice at latency one. This is called a *2Combined-Phased BFS*.

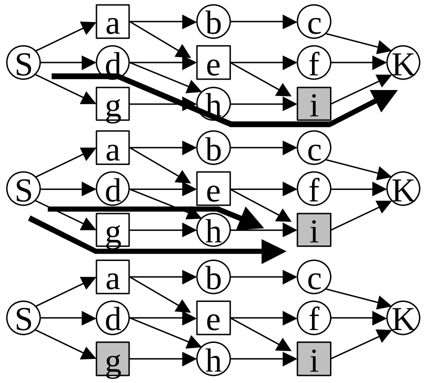

Figure 3. Illustration of the PipeRoute Algorithm.

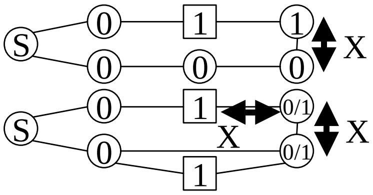

Figure 4. Different Search Styles. Breadth-first search (top) and Combined-Phased BFS (bottom). Notations indicate latency phase of visitation and "X" show deadlocked links.

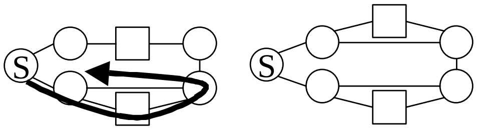

Figure 5. Limitations of PipeRoute. Self-intersection (left) and symmetrical architectures (right).

Unfortunately, their definition of an optimal path allows a route to cross over itself. In the left illustration of Figure 5 we see that the shortest one-latency path is a route that visits a register, then doubles onto itself. However, this is clearly not a valid physical route. The authors justify their definition of an optimal path by indicating that since they use PathFinder in their outer loop, its natural congestion avoidance will resolve these problems over multiple iterations. Unfortunately, we believe that PathFinder may not be able to discourage this type of path self-intersection on many common architectures.

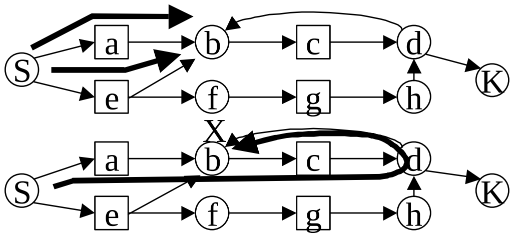

Figure 6. Illustration of QuickRoute Algorithm. Multiple visitation (top) and self-intersection (bottom).

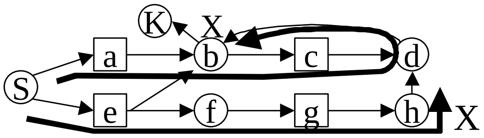

Figure 7. Illustration of QuickRoute Self-Blocking.

First, present sharing cost cannot play a role regardless of architecture design. PathFinder does not update the present sharing of any node until we have found a complete route from a source to sink. Thus, an exploration will not feel the effects of present sharing between the phase zero and phase one routes until we have already completed the search. Furthermore, we cannot update this on-line since a phase one exploration has no efficient way of distinguishing between when it is wrapping back onto itself versus attempting to explore a node that was used at latency zero by an unrelated exploration. Likewise, this problem cannot be resolved by history cost. If we consider a symmetrical architecture as shown in the right illustration of Figure 5, the self-intersecting problem will simply alternate between the top and bottom loop – never realizing that a valid alternative exists.

Because of this characteristic we believe that PipeRoute cannot be used on many FPGA architectures. First, the interconnection flexibility of modern devices will encourage the self-intersecting path problem. That is, high connectivity allows a routing node to easily re-discover itself after going through a pipelining resource. Second, the majority of interconnection networks have a great deal of symmetry. One routing track is likely to have the same access to pipelining resources as neighboring tracks.

4.2 QuickRoute

Instead of gradually assembling higher latency routes from multiple smaller latency segments, QuickRoute attempts to find full N-latency routes directly. Although performed for latencies higher than only zero and one, it is similar to the Combined-Phased BFS from PipeRoute in that we must record the phase of an exploration when a node is visited. In this case, 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 Figure 6, 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, they record the upstream nodes for every exploratory wave and do not allow an exploration to revisit a node already used by itself earlier in the search. In the bottom illustration of Figure 6, a path that goes through *b* will not consider it again for subsequent exploration. This exploration process is simply continued until the sink is discovered at the appropriate latency.

Of course, since the problem is still NP-Complete, this cannot guarantee a solution. For example, if we make a slight modification to the graph, as in Figure 7, we run into problems. If we assume that *k*=1 and we would like to go from *S* to *K* accumulating two registers, we will fail to find a solution. This is because node *b* is initially used by a doomed route that, in turn, prevents the correct route from exploring *d*. Unfortunately, no matter how large we make *k*, we can always construct a graph that will self-block by adding registered paths between *b* and *d*.

However, we believe that QuickRoute still holds multiple advantages over PipeRoute. Not only does QuickRoute defend itself from the self-intersection problem of PipeRoute, it has the flexibility to improve its routing ability by increasing the *k* factor. Furthermore, as we will touch on later, we believe that QuickRoute will generally encounter the self-blocking problem in very specific situations that we can address elsewhere.

5. Cost Formulation and Link Criticality

Unfortunately, neither PipeRoute nor QuickRoute provide delay-driven routing. This might be surprising considering that PathFinder demonstrated how to simultaneously balance congestion and timing over a decade ago. However, there are multiple differences between the conventional routing problem and the pipelined routing problem that prevent us from leveraging PathFinder’s timing-driven cost formulation.

PathFinder takes timing into account by allowing timing-critical nets to follow fast, congested paths while encouraging non-critical nets to seek slower, lower congestion alternatives. In [4], the authors define Eq. 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.

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

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 between source and sink. However, a less critical net will consider both delay and congestion. As A_{ij} approaches zero, the congestion cost will pay a larger role in determining which path is taken.

However, we cannot use this type of timing versus congestion cost formulation to determine pipelined routes. If we compare the N-Delay Routing problem to the conventional routing problem, that there are multiple difficulties in determining the appropriate criticality to use for a given exploration. Primarily, these issues stem from the fact that, in the classical sense, we continuously change the very nature of the netlist during the pipelined routing process.

As seen on the left of Figure 8, the conventional methodology of placing all of the blocks, then routing them produces relatively consistent iteration-to-iteration criticality. In our example, the placement tool has decided that LUT *a* must route to LUT *b* before going to LUT *c*. As routing progresses, Pathfinder can carry over the criticality of the last route found to determine the next route. In this way, PathFinder hinges upon the fact that the routing will not drastically change between iterations. It is unlikely that consecutive routing iterations will choose vastly

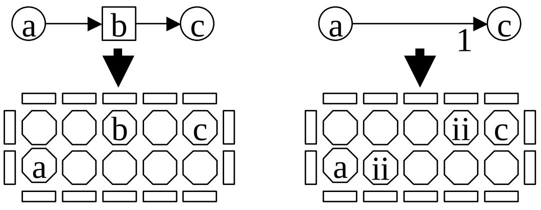

Figure 8. Register Assignment and Criticality. Conventional placement of registers (left) and discovering registers during routing (right). Numbers on edges represent the required latency between source and sink.

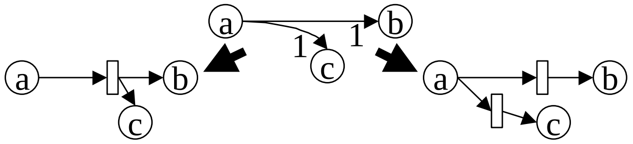

Figure 9. Multi-Terminal Criticality Problem.

faster or slower routes from *a* to *b* or *b* to *c*. However, if this somehow does occur, we will over or under-penalize the congestion versus delay contribution to the overall path cost.

If we consider the same netlist in a pipeline-aware routing framework, as shown on the right of Figure 8, we see that registers are replaced by latency annotations on edges. In this situation, we know that LUT *a* must be connected to LUT *c* by a single latency link. However, the criticality of the individual links between *a* and the register and the register and *c* will heavily depend upon the route that is chosen. The relative criticality of these links will change completely if we choose to register at LUT *i* versus LUT *ii*. This criticality inaccuracy will cause timing oscillation as opposite sides of a register along critical or nearly critical paths vie for dominance.

If the first iteration chooses to register at LUT *i* we can guarantee that the second iteration will choose to register at LUT *ii*, despite that fact that it would be more advantageous, from a timing standpoint, to select either of the middlemost positions. This occurs because the pre-register link will have a very low criticality, making delay on this segment very inexpensive. Conversely, the post-register link will have a very high criticality making delay very costly. If we assume for the moment that there is no congestion in the system, the post-register link will want to become as short as possible at the expense of the pre-register link. Because of this, we will alternately select equally poor register locations and never find the best solution.

Essentially, this type of behavior occurs because the criticality of a link to a register used in one iteration has no bearing if we select a different register during the next iteration. Looking at the problem from a larger scope, we should not be surprised that this occurs. The conventional routing problem only has to contend with between-iteration criticality inaccuracy on a secondary level because the endpoints of all blocks that can affect timing are fixed by the placement before routing begins. If we look at the pipelined routing problem from the standpoint of conventional routing, it is as if we can change the placement of all the registers between every routing iteration.

This problem becomes further complicated if we consider multi-terminal and multi-latency nets. As shown in Figure 9, there are certain situations in which sinks may want to share registers to reduce congestion. However, depending upon their relative placements and if this net becomes critical or near critical, each sink might wish to use a separate register. Unfortunately, it becomes unclear what criticality to assign any of the nets to allow these "zipped" and "unzipped" paths to exist in consecutive iterations and still produce high-quality results. Should the criticality of all latency-*N* segments be averaged? Should the worst criticality of any segment define the criticality of all links? This becomes an issue because we are fundamentally changing the nature of the netlist during routing. Similar to before, in conventional terms it is as if we are performing a limited form of logic synthesis or, at the very least, register duplication between every routing iteration.

6. Determining Link Criticality

Clearly, if we would like to obtain high quality timing results, we cannot use criticality information gleaned from previous routing iterations to guide future exploration. However, PathFinder has shown us that we still need some mechanism to allow more timing-significant links to trade congestion for delay and less important signals to trade delay for congestion. Our solution relies on each exploration to discover its own criticality.

PathFinder manages to route all signals of a circuit in an order-independent fashion by routing each net once disregarding congestion, then ripping up a single net at a time and routing it in the presence of all other nets. While we would normally obtain the timing importance of the signal from the previous routing iteration, we have shown that this cannot be done for pipelined signals. One possible alternative is for an exploration to build its own criticality based upon the delay it has seen thus far. In this scenario, we start with a very low criticality at the source when the exploration has not accumulated any delay and gradually increase the timing significance as the search continues and paths becomes slower. Unfortunately, while this may work for low and mid-criticality links, this will not perform well for high criticality segments. This is because the early portion of all searches will meander to avoid congestion. As the path becomes longer, the search will opt for more direct routes to the sink. Unfortunately for critical nets, the damage has already been done and they will never obtain the congestion-blind routes that they should.

Instead, we suggest that an exploration decides the proper criticality for a route at the only point that the decision can actually be made – when it arrives at the sink. In this formulation, we start *AC* independent waves from the source, each assuming a different criticality from 1/*AC* to 1.0. In this manner, we will have multiple simultaneous searches that each emphasis delay versus congestion in a slightly different way. The first exploration to reach the sink will be the least expensive and, thus, represent approximately the proper balance of congestion versus delay. Furthermore we can trade off runtime for timing accuracy by increasing *AC*. However, if we use this *assumed criticality* methodology in its existing form, we can still suffer from grossly incorrect routing. To understand why, we must return to PathFinder's cost formulation.

Looking at Eq. 1, we can see that high criticality emphasizes low delay and low criticality emphasizes low congestion. Because of this relationship, depending upon the relative values of the architecture's delay and congestion costs, our assumed criticality searches can easily degenerate to always selecting either the lowest or highest assumed criticality for all nets. This is because if the delay values along most paths from the source to the sink are coincidentally larger in magnitude than their congestion counterparts, searches that assume a criticality of 1.0 will always be the cheapest, regardless as to whether they are truly timing critical. A similar situation occurs for the minimum assumed criticality if the relative values are reversed. While this problem could be addressed by ensuring that the delay and congestion values are always balanced, this is not a feasible solution as the congestion values must be able to grow as the routing progresses. Instead, we can solve this issue by using the assumed criticality values to calculate the cost of route up to, but not including, the sink. When we reach a sink, we can re-calculate the cost of the route based upon the actual criticality of the path found.

The complete assumed criticality search methodology, as seen in Figure 10, has several attractive features. First, we have solved the problem of routing inaccuracy due to iteration-to-iteration variance in path criticality. Not only is this clearly an issue for pipelined routing algorithms, this may appear in conventional routing problems on FPGA architectures with particularly heterogeneous routing structures. Second, this approach does not dramatically increase the computational effort of routing.

Obviously, if we conducted *AC* completely independent searches for each source/sink pair, this would only invoke PathFinder's inner loop *AC*-1 additional times. However, we can easily run all of these searches simultaneously and prune non-productive explorations along the way. Of course, once one search has reached the sink, we can end all exploration. However, we can even prune incomplete searches. 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 will be better serviced by the 0.4 assumed criticality exploration. Thus, with the exception of the highest criticality wave, we can prune a search when the current path delay would make the exploration's criticality larger than the next higher assumed criticality search.

7. Timing-Driven Pipeline Routing

Now that we have a methodology to search for routes without a priori knowledge of link criticality, we can incorporate this into the QuickRoute algorithm. To accomplish this, we must first modify the search to assign a routing order to all pipelined nets' sinks. Much like PathFinder's pure Negotiated Congestion algorithm, QuickRoute does not search from the source to any particular sink. Instead, it merely stops when it finds any sink at the proper latency and reinitializes the priority queue with all existing paths. However, to give priority to higher criticality links, we sort each net's sinks first by non-decreasing order of latency (# of registers required on the path), then by non-increasing order of maximum link criticality found in the previous iteration. In this way, the most timing-critical sinks with the fewest chances to amortize path delay over multiple clock cycles determine the earliest stages of the routing tree.

Next, we need to augment our assumed criticality methodology to deal with pipelined routes. In addition to recalculating the true criticality of a link when we discover a sink, we must also do so when we encounter a register. Furthermore, registers must also launch their own series of multi-criticality searches. As shown in Figure 11, if we would like to find a one-latency path between *S* and *K*, we begin at the source with *AC*=3 assumed criticality searches. When one of these waves encounters a register, it recalculates the path cost based upon the real criticality required to reach the register along the given path. When the cheapest path to the register is popped from the priority queue, it launches a new series of *AC*=3 assumed criticality searches of its own at latency one. Notice that although all three zero-latency searches may reach the register and push it into the priority queue, only one path will be deemed the least expensive and, thus, the best way to use this particular register. Only this path will continue on with one-latency explorations.

```
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
```

Figure 10. Assumed Criticality Breadth-First Search.

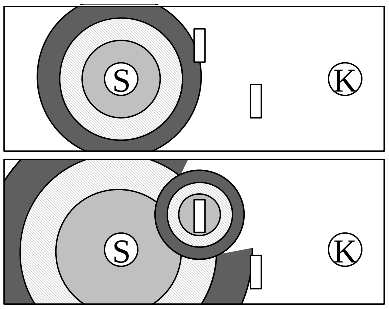

Figure 11. Register-Aware Assumed Criticality Search

Of course, this means that we must define the cost of a multiple latency route. In our example, eventually both registers in the architecture will launch their own set of one-latency explorations. As they near the sink, we need to determine which path best balances not only the congestion and delay of their zero and one-latency paths individually, but the combination of the two. Since each time we encounter a register we can determine the actual criticality of the link, we can define the cost of an L-latency path to be the total of the timing and congestion costs of all zero to L-latency segments, as shown in Eq. 2.

$$C = \sum_{i=0}^{L} (\text{timingCost}_i + \text{congestionCost}_i) \qquad (2)$$

Furthermore, to build successive multi-terminal routes we must also define how pre-existing routes should initialize the priority queue. As seen in Figure 12, after we have found a one-latency route to *K*, we need to push this existing route into the priority queue to reflect all of the possible routing options to the 2-latency sink *J*. While building a link from *b* would allow for the maximum register sharing and will likely cause the minimum congestion impact, developing a wholly new path may offer some timing benefits. Borrowing a concept from timing-driven PathFinder, we consider existing routes to be free in terms of congestion, and we only consider their delay impact on further sinks. Based upon the model discussed in Eq. 2, we push nodes along existing routes into the priority queue by summing only the timing cost of all upstream zero to L-latency segments. For our example in Figure 12, to combine this concept with our assumed criticality methodology, all nodes along *a* would be pushed into the priority queue *AC* times using different assumed criticalities to determine their timing cost. While all nodes along *b* would also be added to the priority queue *AC* times, they would all share some common portion of their cost – the zero-latency timing cost incurred along *a*.

The final modification that we must consider is to the congestion versus timing cost formulation itself. As mentioned in our discussion of potential pitfalls of the assumed criticality methodology, the lowest cost path obtained by using Eq. 1 heavily depends upon the relative values of an architecture's delay and congestion costs. Unfortunately, this will cause further undesirable behavior when we consider routing pipelined paths.

Looking at Figure 13, 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 poor choice because the post-register path is very close to critical and highly congested. If we use Eq. 1 to determine the cost of these two routes, we get Eq. 3 and 4:

$$0.1(d)+0.9(c)+0.9(9d)+0.1(c)=8.2d+c \quad (3)$$

$$0.5(5d)+0.5(5c)+0.5(5d)+0.5(5c)=5d+5c \quad (4)$$

From this, we can see that the selection of balanced versus unbalanced paths also depends upon the relative values of *c* and *d*, an architecture's base delay and congestion cost. In our example, the more balanced path is only selected if $c < 0.8d$. Even if we could somehow guarantee that we correctly scaled the base cost of all routing nodes so that we initially selected more balanced paths, the natural congestion cost escalation of PathFinder will cause later iterations to tend toward worse selections. Not only do these unbalanced paths create a more difficult timing problem, they can actually reverse PathFinder's attempts at congestion resolution. 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 propose making a subtle change that removes this vulnerability. If we divide Eq. 1 through by $(1-A_{ij})$, we obtain Eq. 5:

$$\frac{C_n}{1-A_{ij}}=\frac{A_{ij}d_n}{1-A_{ij}}+\frac{(1-A_{ij})c_n}{1-A_{ij}}=\frac{A_{ij}}{1-A_{ij}}d_n+c_n \quad (5)$$

While this does not change path selection for conventional non-pipelined routing, since all routes will evenly scaled by $(1\text{-}A)^{-1}$, this does change the behavior for pipelined signals. If we revisit our example from Figure 13 and substitute the new cost formulation we get Eq. 6 and 7:

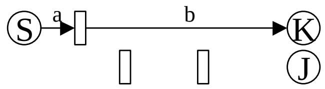

Figure 12. Re-initializing PQ for Multi-Terminal Nets

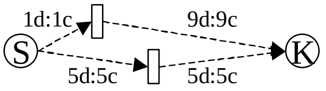

Figure 13. Congestion Versus Timing Cost Dependency of PathFinder Cost Formulation. Notation is (delay:congestion) and we assume a circuit-wide critical path of 10d.

```
Armada(net, numAC, maxK,critPath)
for all nodes N in architecture, for all latencies L, for all assumed
     criticalities CFac, clear n.visited[L][CFac]
sort net.sinks by non-decreasing latency, non-increasing criticality
insert source into net.routingTree
for all sinks K in net.sinks
    Initialize PQ with existing routes
     for all nodes N in net.routingTree, for CFac = 1/numAC to 0.99
        if CFac != 0.99 && N.delay>(CFac +1/numAC)*critPath
            continue
        else
            insert N into PQ at N.timingCost(CFac), latency N.latency
        end if
    end for
    Search for L-latency route to sink
    while !PQ.empty
        remove cheapest node N, latency L, assumed crit. CFac from PQ
        if N == K && L == K.latency
            add route to net.routingTree, empty PQ, clear all n.visited
            continue next sink
        else if N.visited[L][CFac] == maxK, continue
        else
            N.visited[L][CFac]++
            for each neighbor X of N
                if X.visited[L][CFac] < maxK
                    if X != K && X != register
                        if CFac != 0.99 && X.delay>(CFac +1/numAC)*critPath
                            continue
                        else
                            push X into PQ at X.timingCongestionCost(CFac), latency L
                        end if
                    else
                        calculate actual crit CR to X
                        if X == register && K.latency <= L + 1
                            push X into PQ at X.timingCongestionCost(CR), latency L + 1
                        else
                            push X into PQ at X.timingCongestionCost(CR), latency L
                        end if
                    end if
                end if
            end for
        end if
    end while
    return failure
end for
```

Figure 14. Armada Timing-Driven Pipeline Net Router.

$$0.11(d)+c+9(9d)+9c=81.11d+10c \quad (6)$$

$$1(5d)+5c+1(5d)+5c=10d+10c \quad (7)$$

Since both the congestion and delay costs are necessarily positive numbers, we can see that more balanced paths are now selected over unbalanced paths without the need to meticulously adjust the relative values of an architecture's base congestion and delay costs. However, the router still has the option of selecting the unbalanced path should it become less congested in future routing iterations.

Now, we can see that the cost of an entire L-latency path becomes the summation of the congestion encountered plus the relative delay of each link adjusted by its individual criticality. This ensures that a link with high criticality will not be able to mask high congestion.

$$C=\sum_{i=0}^{L}\text{timingCost}_i+\sum_{i=0}^{L}\text{congestionCost}_i \quad (8)$$

If we combine all of these techniques we get the Armada timing-driven pipeline routing algorithm. It begins with PathFinder's basic timing-driven congestion resolution engine in an outer loop. Taking a suggestion from [1], we saturate link criticality at 0.99. However, as shown in Figure 14, we then depart from the conventional routing problem entirely by adding in a modified

QuickRoute inner loop that has been enhanced with our timing-driven sink ordering, assumed criticality methodology and new timing-driven cost formulation.

8. Results

To determine the effectiveness of the Armada algorithm, we follow PipeRoute and QuickRoute's lead and test our codebase within the RaPiD framework. Our evaluation encompasses nine RaPiD netlists, shown in Table 1, with a wide range of pipeline requirements. These are mapped to three different RaPiD architectures: the original architecture that contains 16 logic blocks per cell, length-4 short tracks, length-16 long tracks, and three optional registers at each bus connector, and two other architectures that are similar, but substitute long tracks of length 4 and 8. These modified architectures allow us to test harder timing-driven routing problems by increasing the number of registering options.

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 algorithms. While conventional placement always attempts to group interconnected blocks as closely as possible, this is not necessarily favorable on pipelined architectures. This is because, as mentioned earlier, high latency connections may need to take a circuitous route if there are not enough pipelining resources between the logic blocks to acquire the appropriate registering.

For the PipeRoute router, we use the version from [5] that augmented the original PipeRoute algorithm with a rudimentary timing-driven formulation. In the new methodology, the maximum criticality encountered by any link between a given source and sink determines the overall net criticality during the following routing iteration. For example, if a given three-latency pipelined signal is connected from the source to the sink by four segments of delay (1, 1, 1, 3), the criticality passed on to PathFinder's timing-driven cost formulation will be three divided by the critical path delay for all PipeRoute explorations between the source/sink pair. Of course, this introduces some inaccuracies into the system. Not only does this methodology suffer from the relative cost interrelationship between congestion and timing that inspired our modified cost formulation, it also suffers from the false link criticality predictions that we addressed with our assumed criticality approach.

Testing began with the original RaPiD architecture. We first performed six independent PipeRoute placement and routing runs and gathered the best results. The placements from these tests were then routed using congestion-driven QuickRoute, the Armada algorithm, and the Armada algorithm substituting in the original PathFinder cost formulation. Although we followed [3]'s suggestion of k=1 for both QuickRoute and Armada, we arbitrarily set the number of assumed criticality searches for both Armada runs to AC=10.

In Table 2-4, *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 PR* results are obtained when each tool is given the same number of tracks that PipeRoute requires for a given netlist. *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. *Match Tracks* results do not include results for PipeRoute as the provided codebase does not allow the placement and routing steps to be separated. Given a different architecture, PipeRoute will also change the placement.

Table 1. RaPiD Netlist Characteristics. Min Registers is the minimum number of registers a netlist needs assuming maximum register sharing. Max Latency refers to the largest pipelining depth for a single sink.

Netlist	RaPiD Cells	Min Registers	Max Latency
firtm	16	20	16
fft16	12	40	3
sobel	18	49	5
matmult4	16	129	31
imagerapid	14	149	11
sort_rb	11	159	31
sort_g	11	159	32
cascade	16	226	21
firsymeven	16	377	31

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. All results were gathered on 3.2GHz Intel Xeon machines with 2GB of RAM. Unfortunately, runtime is only reported for the three QuickRoute-derivative routers because differences in code execution prevented meaningful comparisons to be made with the PipeRoute codebase. This said, we generally expect the original QuickRoute algorithm to perform as fast or faster than PipeRoute since it does not perform multiple piecewise searches.

As seen in Table 2, our first surprise is that the original congestion-driven QuickRoute algorithm actually performs nearly as well as the improved timing-driven PipeRoute formulation. Although based upon the results in [3] we would expect QuickRoute to provide marginally better track counts, we believe that the similar timing results indicate that the timing-driven PipeRoute formulation is largely ineffective. As predicted, it is likely that inaccuracies within the timing-driven formulation itself greatly limit the ability for optimization.

In contrast, the Armada algorithm finds vastly superior timing results with essentially identical routability. Clearly, Armada is able to improve timing by roughly 50-70% over previous approaches – even occasionally using fewer tracks. This is likely because the timing-driven cost formulation provides additional direction to the QuickRoute-like searches avoiding some occurrences of self-blocking. As expected given the AC=10 factor, Armada runs approximately 10x slower than QuickRoute. We can also see that our new timing-driven cost formulation functions as intended when we substitute PathFinder's into the Armada algorithm (*Armada-PFind*). We can see that this alone is responsible for at least a 15% improvement in critical path, not to mention much faster routing convergence.

As seen in Table 3 and Table 4, this trend continues if we migrate to more difficult routing problems. We repeated the testing methodology used on the original RaPiD architecture on architectures with double and quadruple the number of pipelined switch opportunities. First, we can see that the problem becomes more difficult from the standpoint of congestion resolution. Long

Table 2. Normalized Results for Original RaPiD Architecture. All results normalized to the best Armada run.

Best Track	Tracks	Crit. Path	Runtime	Ave. Runtime
	1.09	1.56	-	-
QuickRoute	1.04	1.64	**0.10**	**133 s**
Armada	**1.00**	**1.00**	1.00	1721 s
Armada-PFind	1.03	1.18	7.65	7982 s
	1.09	1.56	-	-
QuickRoute	1.09	1.75	**0.08**	**113 s**
Armada	1.09	**1.00**	0.94	1752 s
Armada-PFind	1.09	1.19	5.21	5329 s
	1.05	1.73		
Armada	1.05	**0.99**	1.05	1826 s
Armada-PFind	1.05	1.20	7.45	7705 s

Table 3. Normalized Results for Length-8 Long Tracks. All results normalized to the best Armada run.

Best Track	Tracks	Crit. Path	Runtime	Ave. Runtime
	1.00	1.66	-	-
QuickRoute	**0.96**	1.65	**0.10**	**66 s**
Armada	1.00	**1.00**	1.00	1357 s
Armada-PFind	1.02	1.30	3.11	3068 s
	1.03	1.71		
Armada	1.03	**1.00**	0.88	841 s
Armada-Pfind	1.03	1.31	3.12	3075 s

Table 4. Normalized Results for Length-4 Long Tracks. All results normalized to the best Armada run.

Best Track	Tracks	Crit. Path	Runtime	Ave. Runtime
	1.01	1.59	-	-
QuickRoute	1.02	1.54	**0.11**	**76 s**
Armada	**1.00**	**1.00**	1.00	2637 s
Armada-PFind	1.05	1.21	2.75	2976 s
	1.05	1.55		
Armada	1.05	**0.99**	0.84	1593 s
Armada-PFind	1.05	1.21	2.75	2976 s

tracks have been split into multiple independent segments, so although the routing can be compressed into fewer tracks, contention for these resources becomes fiercer. This is shown by the fact that the track gap between PipeRoute and the QuickRoute-derivatives mostly closed. This is also the reason that we no longer show *Match PR* results.

Although we have proven that Armada can obtain significantly better pipelined routing results than any of its predecessors, there are still two outstanding questions regarding its effectiveness. First, as mentioned earlier, the maximum visitation factor that we used in our testing (*k*=1) was suggested by the original QuickRoute paper. Even though we are operating within the same architectural framework, the timing-driven nature of our problem formulation might make more thorough explorations attractive. As seen in Table 5, there is a correlation between larger values of *k* and higher quality results, but the change is relatively minor. In general, we expect this behaviour to be highly architecture-specific.

Second, we completely arbitrarily chose the number of assumed criticality searches that we would use (*AC*=10). Since the assumed criticality completely controls how paths weigh congestion versus delay for the majority of a given route, we expect the quality of the timing to heavily depend upon the granularity of our assumed criticality factors. However, looking at Table 6, we see that although we get a marked runtime improvement, dramatically decreasing the number of assumed criticality searches does not necessarily affect the overall quality of the routing. In fact, we do not really see a decline in quality even if we reduce the number of searches to merely two (only assume criticalities of 0.5 and 0.99) unless we migrate from the original architecture to systems with shorter long-distance tracks, and therefore more registered switchpoints.

Table 5. Normalized Results for Armada, *k*=1, 2, 4. All results normalized to k=1 values

16-Length	Tracks	Crit. Path	Runtime	Ave. Runtime
	1.00	1.00	1.00	1721 s
k = 2	0.99	0.95	1.67	2454 s
k = 4	0.98	1.02	3.24	5634 s
	1.00	1.00	1.00	1357 s
k = 2	0.99	1.00	1.29	1757 s
k = 4	0.98	1.00	6.48	21725 s
	1.00	1.00	1.00	2637 s
k = 2	1.01	1.00	1.77	4414 s
k = 4	1.00	0.98	3.93	9452 s

Table 6. Normalized Results for Armada, *AC*=10, 8, 6, 4, 2. All results normalized to AC = 10 values

16-Length	Tracks	Crit. Path	Runtime	Ave. Runtime
	1.00	1.00	1.00	1721 s
AC = 8	0.98	1.04	0.96	1168 s
AC = 6	1.00	0.97	0.67	785 s
AC = 4	0.99	1.02	0.52	573 s
AC = 2	1.00	0.97	0.36	354 s
AC = 1	1.11	1.20	0.35	300 s
	1.00	1.00	1.00	1357 s
AC = 8	0.98	1.01	0.79	910 s
AC = 6	0.98	1.01	0.43	576 s
AC = 4	1.00	1.01	0.56	692 s
AC = 2	1.00	1.04	0.29	307 s
AC = 1	1.08	1.37	0.32	217 s
	1.00	1.00	1.00	2637 s
AC = 8	1.01	0.99	0.75	1802 s
AC = 6	0.99	1.00	1.05	1803 s
AC = 4	0.99	0.98	0.60	1014 s
AC = 2	1.04	1.01	0.31	342 s
AC = 1	1.60	1.51	0.67	1080 s

Although this may seem counter-intuitive, looking at the routed results found by Armada we see this is actually an artifact of the original RaPiD design philosophy. In almost all cases, the critical path reaches some architectural limit – two to three bus connector-to-bus connector delays or less. If we consider that RaPiD was built to be a deeply pipelined architecture, this should not be particularly surprising. In this case, the router is merely finding exactly the types of routes that the original designers had anticipated. If the router achieves such an extremely low critical path delay, all signals actually become either 50% or 100% critical making *AC*=2 work exceedingly well. It is only when we shrink the length of long tracks considerably and add a huge number of registers that we begin to produce paths that do not register at nearly every switchpoint they traverse. However, as with determining *k*, *AC* behaviour is also likely highly architecture dependent. The majority of systems do not have the extremely predictable routing characteristics of the RaPiD architecture and we expect more conventional pipelined FPGAs to be more sensitive to the number of assumed criticality searches.

9. Future Work

Although Armada performed very well on the RaPiD architecture, the experiments outlined in this paper relied heavily on the existing PipeRoute codebase. Other FPGA architectures, like the pipelined island-style devices proposed by Singh and Brown, raise a serious unanswered question for a pipeline-aware CAD toolflow as a whole. While the Armada router itself operates on an architecture-independent graph, the quality of results depends wholly upon the quality of the placement. Although the placements obtained for our experiments were certainly sufficient, the pipeline-aware placement problem itself is largely unanswered.

PipeRoute addresses pipeline requirements during its placement phase by first assuming that all pipelined sinks for a given net will share as many registers as possible. To calculate the quality of a given placement, it temporarily binds these registers to real locations in the architecture. While this appears to function adequately for RaPiD, this will likely be too rudimentary for more flexible or complex architectures. Not only does RaPiD's 1-D nature make the placement problem already very simple, RaPiD has a very regular, predictable and pipeline resource-rich routing structure. Thus, any inaccurate assumptions made in the placement phase can likely be easily compensated for during the routing phase.

However, we expect that better placement algorithms will be necessary for the majority of other pipelined architectures in order to make the pipelined routing problem tractable. Not only does the 2-D nature of island-style FPGAs automatically make the routing problem more difficult, we believe that errors in the placement can dramatically increase the likelihood that QuickRoute-derivative searches will run into problems with self-blocking sub-optimality.

10. Conclusions

Although we have shown in this paper that a timing-driven solution to the N-Delay Routing problem is necessary to build fast and efficient pipelined FPGAs, previous solutions have serious limitations. At the expense of both system area and timing, early pipelined architectures avoided the problem altogether by constraining their retiming tools and the fundamental design of the architectures themselves. Although later attempts tackled the N-Delay Routing problem head-on, these had several shortcomings including limited routability on modern FPGA architectures and poor timing performance.

Primarily, the timing-driven N-Delay Routing problem is difficult because, from the viewpoint of conventional CAD tools, it contains aspects of both cell placement and register duplication that must be solved simultaneously with the normal routing problem. 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 routers, we combined these to form the Armada timing-driven pipeline-aware routing algorithm. On three RaPiD architectures this algorithm was shown to provide roughly 60% better average timing results without dramatically affecting routability. While more computationally intensive than previous algorithms, Armada remains competitive, especially given the dramatic timing benefits.

Although there are still some unanswered questions regarding a complete timing-driven pipeline-aware FPGA CAD tool suite, we believe that the Armada routing algorithm is the first step towards developing modern pipelined reconfigurable computing devices.

11. References

[1] Betz, Vaughn, Jonathan Rose, and Alexander Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.

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

[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. “Integrated Retiming and Placement for Field Programmable Gate Arrays”. *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2002: 67-76.

[9] 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.

[10] 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.