AIR: A Fast but Lazy Timing-Driven FPGA Router

Kevin E. Murray, Sheng Zhong, and Vaughn Betz
Department of Electrical & Computer Engineering
University of Toronto, Canada
{kmurray,vaughn}@ece.utoronto.ca

Abstract— Routing is a key step in the FPGA design process, which significantly impacts design implementation quality. Routing is also very time-consuming, and can scale poorly to very large designs. This paper describes the Adaptive Incremental Router (AIR), a high-performance timing-driven FPGA router. AIR dynamically adapts to the routing problem, which it solves 'lazily' to minimize work. Compared to the widely used VPR 7 router, AIR significantly reduces route-time (7.1× faster), while also improving quality (15% wirelength, and 18% critical path delay reductions). We also show how these techniques enable efficient incremental improvement of existing routing.

I. Introduction

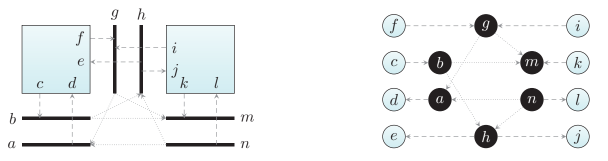
Field Programmable Gate Arrays (FPGAs) have pre-fabricated routing resources along with configurable switches to interconnect them. This allows the routing resources to be re-configured to implement different designs, but also means the interconnect network has limited and restricted connectivity. The interconnect network typically accounts for a majority of delay, and 50%+ device area [1], [2]. As a result the routing of signals through the network has a large affect on the final design implementation quality (wirelength, power consumption and critical path delay). Unlike Application Specific Integrated Circuits (ASICs) where custom interconnect topologies can be constructed, FPGA routing requires finding a legal *embedding* of the design's interconnect into the pre-fabricated network. Routing also accounts for a large portion of FPGA CAD flow run-time (41-86% in commercial and academic tools [3]) and can scale poorly to very large designs due to high-fanout nets and difficult to resolve congestion.

In the FPGA CAD flow there are no stages following routing which can fix up routability or timing issues (e.g. no buffer insertion or gate re-sizing). The only way for designers to address these issues is to restructure their design's logic (to reduce congestion and/or improve timing), which is extremely disruptive. It is therefore very important for FPGA routers to be robust (find legal routings) and high quality (minimize wirelength and delay).

An FPGA's routing resources and switches are typically modelled as a Routing Resource (RR) graph as shown in Figure 1, where conductors (wires/pins) and switches correspond to nodes and edges respectively. The FPGA routing problem is to find a large number of non-overlapping trees within the RR graph which implement the design's connectivity; while simultaneously minimizing metrics such as wirelength and critical path delay, all within reasonable run-time. This is a challenging task given the large size of the RR graph[1] and its limited connectivity (sparseness).

The most successful FPGA routing techniques are based upon the PathFinder negotiated congestion algorithm [4],

[1]Tens to hundreds of millions of nodes for modern FPGAs



(a) Routing Architecture (b) Routing Resource Graph

Fig. 1: Routing Resource Graph. Dotted and dashed arrows represent configurable switches.

where multiple nets are allowed to use the same routing resources. Such overused resources are said to be *congested*. Allowing congestion prevents nets from blocking each other, an important characteristic given the interconnect network's limited flexibility. To resolve congestion and arrive at a legal solution nets are repeatedly ripped-up and re-routed while modifying routing resource costs.

A variety of works have studied FPGA routing. Early works focused on two-stage global-detailed routing [5] and investigated heuristics for Steiner point selection to improve quality [6]. However both were outperformed by single-stage negotiated congestion routing [7]. Different criteria for ranking paths during graph search were explored in [8], improving quality at the cost of significantly higher run-time. [9] investigated methods to prune the search space and build good initial 'spines' for high-fanout nets. More recently, CRoute [10] proposed decomposing nets into independent connections to reduce route-time. Unlike CRoute, our approach maintains the natural net-based structure of the problem which improves routing resource re-use, reduces redundant work during graph search, and facilitates further net-based optimizations.

Another popular approach has been to exploit parallel computing resources to speed-up routing [11], [12], [13], [14]. However these techniques have also faced scalability challenges due to synchronization costs and load imbalances. Our approach is complementary as it reduces the work required to route individual nets, and particularly large high-fanout nets which often limit parallel speed-up [11].

We focus on developing the Adaptive Incremental Router (AIR) which improves scalability and quality compared to previous approaches. Our contributions include:

- 'Lazy' methods to improve routing scalability for large designs and high-fanout nets (Section II),
- Techniques to adapt to the routing architecture and design characteristics to improve quality and robustness (Section III),
- An approach to efficiently improve the quality of an existing routing (Section IV), and
- A comparison of our approach to several academic [15], [10] and commercial [16] routers (Section V).

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

II. Lazy Routing

Our routing algorithm is based upon PathFinder [4] as implemented in VPR [15]. In large designs routing has been found to scale poorly [3] due to the difficulty of resolving congestion and routing high-fanout nets. To improve these characteristics we focus on making the router as 'lazy' as possible by avoiding unnecessary work.

Algorithm 1 outlines AIR's netlist routing algorithm. The algorithm operates over multiple *routing iterations* (Line 4). During each iteration, connections are routed between each net's driver and sinks (Lines 6 and 7), potentially sharing routing resources with other nets. Once all nets have been routed (Line 5), the routing resource costs are selectively increased based on present and historical congestion (Line 14), with the aim of reducing congestion during subsequent iterations. This repeats until a legal routing is found (Lines 8 and 11), or the design is deemed unroutable by hitting the iteration limit (Line 4). Finally, the best legal routing found (if any) is returned (Line 18).

Algorithm 1 AIR Netlist Router

```
Require: $nets$ to route, $\alpha$ the maximum number of convergences
Returns: $best$ routing found
 1: function AIR_ROUTE($nets, \alpha$)
 2:   $best \leftarrow \emptyset, curr \leftarrow \emptyset$          ▷ Best and current route trees for nets
 3:   $convergences \leftarrow 0$                  ▷ Number of legal routings found
 4:   for $iter \in 1 \ldots max\_iters$ do
 5:     for $net \in nets$ do
 6:       for $sink \in$ UNROUTED_SINKS($net, curr[net]$) do    ▷ Incr. route
 7:         $curr[net] \leftarrow$ AIR_ROUTE_CONNECTION($net, curr[net], sink$)
 8:     if IS_LEGAL($curr$) then
 9:       $best \leftarrow$ BEST_ROUTING($best, curr$)        ▷ Keep best routing
10:       $convergences \leftarrow convergences + 1$
11:       if $convergences = \alpha$ then          ▷ Convergence limit reached
12:         break
13:       RESET_PRES_FAC()                 ▷ Reduce present congestion cost
14:     UPDATE_COSTS()                        ▷ Update pathfinder costs
15:     for $net \in nets$ do              ▷ Prepare for incremental re-route
16:       RIPUP_ILLEGAL_CONNECTIONS($current[net]$)
17:       RIPUP_DELAY_DEGRADED_CONNECTIONS($current[net]$)
18:   return $best$
```

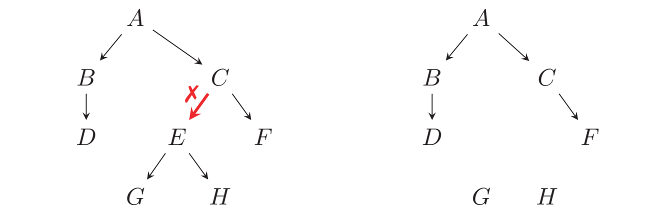
A. Incremental Routing

In PathFinder-based routers each net's routing is represented as a *route tree*: a tree of routing resources spanning from the net's driver (tree root) to each of its sinks (tree leaves). Each path in the route tree (from root to leaf) corresponds to a net connection (from driver to sink). To reduce wirelength it is usually desirable for connections within the same net to share some routing resources.

Traditional PathFinder-based routing algorithms rip-up and re-route all nets every routing iteration. However in practise many nets (or portions of nets) may have already been legally routed. In such cases it is not strictly necessary to re-route these connections. AIR exploits this by re-routing nets incrementally on a per-connection basis. This 'lazy' approach avoids redundantly ripping up (and then re-routing) the legal portions of a net.[2]

First, a net's current route-tree is walked to identify congested sub-trees as shown in Figure 2a. Illegal sub-trees are then pruned as shown in Figure 2b to leave only legal routing

[2]This is a much more fine-grained approach than [12], which only avoided ripping-up completely legal nets.



(a) Connection $C \rightarrow E$ is illegal (used by another net).
(b) Pruned route tree. Sinks G and H must be re-routed.

Fig. 2: Pruning of route tree connecting A to $\{D, G, H, F\}$

TABLE I: Lazy Routing Evaluation on Titan Benchmarks (Normalized Geomean)

	Routed WL	Crit. Path Delay	Route Time
baseline	1.00	1.00	1.00
incremental ($\delta = 16$)	0.99	1.00	0.75
high_fanout ($\beta = 64$ $\gamma = 1.0$)	1.00	1.03	0.75
high_fanout ($\beta = 64$ $\gamma = 0.9$)	1.00	1.02	0.79
both ($\delta = 16$ $\beta = 64$ $\gamma = 1.0$)	0.99	1.02	0.55
both ($\delta = 16$ $\beta = 64$ $\gamma = 0.9$)	0.99	1.01	0.60

(Algorithm 1 Line 16). Second, the pruned net sinks are re-routed during the next routing iteration (Algorithm 1 Line 6), using the net's remaining legal routing as potential branch points.[3]

While incremental routing is primarily a run-time optimization it can impact Quality of Results (QoR). In particular, since PathFinder uses a present congestion cost, not ripping up all connections may cause some timing critical nets to take indirect routes to avoid inducing congestion. To alleviate this we perform targeted delay-based rip-up (Algorithm 1 Line 17), which forces timing-critical connections whose delay has degraded (compared to previous iterations) to be re-routed even if their routing was legal. Since these ripped-up connections are timing critical they are re-routed with a focus on delay in the following iteration, preventing the router from converging to a poor critical path delay solution. Since only a small number of critical connections are ripped-up, routing still converges quickly to a legal solution.

For small nets the benefit of incremental routing is minimal, so we apply incremental net re-routing only for nets beyond a certain fanout threshold (δ). We empirically found $\delta = 16$ performed best, although values between 1 and 512 all performed well.

Table I compares the impact of `incremental` routing to the `baseline` (where nets are always ripped up) on the Titan benchmarks [3]. The results show that incremental routing slightly improves wirelength and has no impact on critical path delay, while reducing router run-time by 25%. Notably, the run-time benefit is often more significant on large circuits, with the largest benchmark (gaussianblur) completing 2.0× faster.

B. High-Fanout Routing

High-fanout nets, which often span a large portion of the device, are particularly time-consuming to route. AIR routes nets one connection at a time (Algorithm 1 Line 7) using Algorithm 2. To avoid wasting wiring, existing routing (from a net's previously routed connections) is added to the

[3]Note the router already stores route trees for each net, so no additional memory is required for incremental routing.

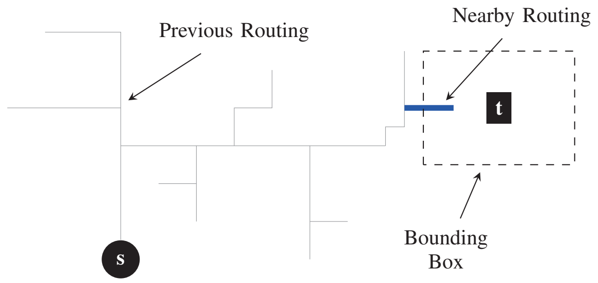


Fig. 3: High fanout routing from driver (s) to sink (t).

heap, allowing it be re-used as a branch-point for subsequent connections (Algorithm 2 Line 7).

Algorithm 2 AIR Connection Router

```
Require: $net$ being routed, existing $route\_tree$, target $sink$
Returns: updated $route\_tree$ with branch to $sink$
 1: function AIR_ROUTE_CONNECTION($net, route\_tree, sink$)
 2:   $path \leftarrow \varnothing$
 3:   if FANOUT($net$) $\geq \beta$ and CRITICALITY($sink$) $< \gamma$ then
 4:     $heap \leftarrow$ INIT_NEARBY_ROUTING($route\_tree, sink$)
 5:     $path \leftarrow$ FIND_PATH_FROM_HEAP($heap, sink$)
 6:   if $path = \varnothing$ then
 7:     $heap \leftarrow$ INIT_FULL_ROUTE_TREE($route\_tree$)
 8:     $path \leftarrow$ FIND_PATH_FROM_HEAP($heap, sink$)
 9:   UPDATE_ROUTE_TREE($route\_tree, path$)
10:   return $route\_tree$
```

Most designs have a small number of high-fanout nets, but we found these few nets typically consume 12-34% of run-time on the Titan benchmarks. For a high-fanout net with k sinks placing the entire route tree into the heap causes the router's time complexity to grow as $O(k^2 \log k)$, since the $O(k)$ routing must be added to the heap[4] for each of the k sinks. In practise, as shown in Figure 3, branching from parts of the high-fanout net which are far from the target sink are unlikely to lead to a lower cost path. Instead of pushing such potential branch points into the heap, AIR uses an approach derived from [17] and lazily puts only existing routing which is spatially near the target sink into the heap (Algorithm 2 Line 4). This reduces the complexity of routing a high-fanout net to $O(k)$.[5] We also exploiting this spatial information to limit the router's search space to a region which contains both the previous routing and the target sink.

Unlike the FPGA routing architectures considered in [17], modern FPGAs use uni-directional wires of differing wire-lengths and varying connectivity for area efficiency and speed [18]. These differences mean in some cases no path can be found from the spatially nearby previous routing to the target sink.[6] In such cases we fall back to placing the full route tree onto the heap (Algorithm 2 Lines 6 to 8). Since such instances are rare, this maintains run-time efficiency in the typical case while ensuring robustness.

Furthermore, unlike [17] (which only considered routability), we observed some degradation in timing performance, since high-fanout timing-critical connections will have less flexibility to find low delay routes. For each connection we calculate its *criticality*: the ratio of connection slack to the associated timing constraint [19]. We then route all timing critical high-fanout connections (those with criticality $> \gamma$) using the full previous route tree (Algorithm 2 Line 3).

[4]$O(k \log k)$ work

[5]Only a constant number of RRs are added to the heap per sink.

[6]For instance if the spatially nearby wire has limited connectivity to block pins and is heading in the wrong direction.

TABLE II: Impact of Router Lookahead and Base Costs on Titan benchmarks (Normalized Geomean)

	Routed WL	Crit. Path Delay	Route Time	Peak Memory
classic	1.00	1.00	1.00	1.00
map	0.98	0.91	5.94	1.00
classic_length	0.93	0.99	0.76	1.00
map_length	0.92	0.91	0.89	1.00

Table I also shows the impact of high fanout routing. Compared to the `baseline`, high fanout routing (`high_fanout` $\beta = 64$ $\gamma = 1.0$) reduced run-time by 25% while slightly degrading critical path delay. On large designs the run-time improvement can be more dramatic (up to 2.0$\times$ on directrf). Setting an active criticality threshold ($\gamma < 1$) improves delay, at the cost of a small reduction in run-time improvement.

Finally, Table I shows the impact of both lazy routing techniques together (`both`); run-time is reduced by 40% (up to 5.2$\times$ on directrf). These results show that AIR's lazy routing approach significantly improves run-time and yields more significant improvements on larger designs, showing its enhanced scalability.

III. Adaptive Routing for Quality & Robustness

In addition to routing lazily to minimize work, AIR also adapts to the nature of the routing problem by creating a routing architecture aware lookahead and dynamically adjusting how it searches for paths based on congestion.

A. Router Lookahead

AIR's connection router (Algorithm 2 Lines 5 and 8) uses an A*-like search algorithm [20], with a predictive lookahead[7] to estimates the cost (delay and congestion) of reaching the target sink through the current node being explored. This guides the router to quickly find a low cost path to the target. The VPR 7 router lookahead (which we call the *classic* lookahead) makes simplifying assumptions which may not hold true on modern FPGA architectures, like the Stratix IV-like architecture used in this section. In particular, it assumes different wire types (e.g. wire lengths) do not interconnect, and all wire types connect to block pins. The classic lookahead can therefore mislead the router on modern architectures, harming delay and wirelength.

AIR uses a new lookahead based on an enhanced version of [18], which adapts to the targeted routing architecture. The new lookahead uses an undirected Djikstra-based search to quickly profile a large number of different routes through the RR graph. As shown in Algorithm 3, this search is performed for a handful of sample locations, and every wire type and orientation (vertical or horizontal). This produces the delay and congestion costs to travel different horizontal and vertical distances. This data is reduced into a concise *map* of the routing network by making the common assumption of translational invariance in the FPGA routing network, so only differences in position need to be stored.

The first two rows of Table II show the impact of the different lookaheads on the Titan benchmarks. First, compared to the `classic` lookahead, the new `map` lookahead achieves much better quality; reducing critical path delay by 9% and

[7]Akin to a heuristic lower bound, but not strictly admissible.

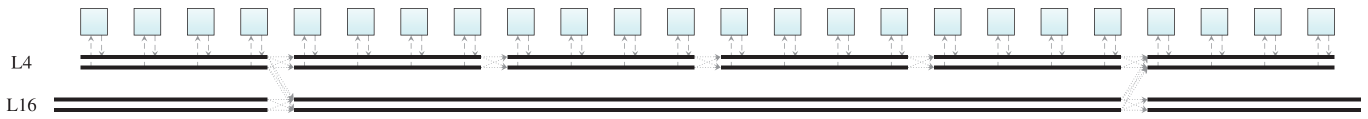


Fig. 4: Stratix IV-like hierarchical routing architecture with short (L4) and long (L16) wires. Long wires are only accessible via short wires, and do not directly connect to block pins.

Algorithm 3 Map Lookahead Creation

```
1:  function BUILD_MAP_LOOKAHEAD( )
2:    $map \leftarrow$ INIT_MAP_INF()                          ▷ Initialize all entries to $\infty$
3:    for $loc \in SAMPLE\_LOCS$ do                          ▷ Each sample location
4:      for $w \in WIRE\_TYPES$ do                          ▷ Each wire type (e.g. length)
5:        for $c \in \{CHANX, CHANY\}$ do                    ▷ Each wire orientation
6:          $ref\_node \leftarrow$ PICK_REF_NODE($loc, w, c$)
7:          $costs \leftarrow$ DJIKSTRA_FLOOD_FROM($ref\_node$)
8:          for $x \in 1 \ldots W$ do                           ▷ Each horizontal position
9:            for $y \in 1 \ldots H$ do                         ▷ Each vertical position
10:             $dx, dy \leftarrow |loc.x - x|, |loc.y - y|$          ▷ Position difference
11:             $delay \leftarrow$ MIN($costs[x][y].delay, map[w][c][dx][dy].delay$)
12:             $cong \leftarrow$ MIN($costs[x][y].cong, map[w][c][dx][dy].cong$)
13:             $map[w][c][dx][dy].delay \leftarrow delay$
14:             $map[w][c][dx][dy].cong \leftarrow cong$
15:   return $map$
```

wirelength by 2%, and does not increase the peak memory footprint. However route-time increases by nearly 6×. The high route-time will be alleviated by adjusting the base costs as discussed in the following section.

Critical path delay improves since the map lookahead's profiling captures the effects of hierarchy in the FPGA interconnect network. As illustrated in Figure 4, modern FPGAs often have high-speed long wire sub-networks which are only accessible from a subset of the more plentiful short wires [21], [2]. Figure 5 shows the delay estimates produced by the lookaheads for various distances starting from a short wire when targeting a Stratix IV-like architecture similar to Figure 4. In Figure 5a the classic lookahead assumes all connections use the same type of short wire, leading delay to increase rapidly with distance. In contrast, Figure 5b shows the map lookahead's delay estimate increases much more slowly, particularly at longer distances. The map lookahead captures the effect of using fast long wires to traverse long distances (even when starting from a short wire).

These differences guide the router to make different choices for timing critical long distance connections. The classic lookahead guides the router to immediately drive towards the target using the short-wire network (since it does not understand faster paths may exist). The map lookahead instead guides the router to search the short-wire network more thoroughly to find a way onto the faster long wire network early, improving delay over long distances.

B. Wire Base Costs

The lookaheads also provide congestion/resource cost estimates to guide the router's search process. The congestion cost estimates produced by the different lookaheads for various distances are shown in Figure 6. Figures 6a and 6b show the classic lookahead's congestion cost estimates for short and long wires respectively. Interestingly, these shows it is much cheaper to travel an equivalent distance using long rather than short wires – even though the long wires are much larger and rarer routing resources.[8] This cost difference biases the router to prefer using the long wire network even for non-timing-critical connections. Since the map lookahead also has a stronger preference for using long wires for timing-critical connections this leads to significant congestion in the long wire network. While this congestion will eventually resolve through congestion negotiation, it is a slow process: requiring connections to be repeatedly ripped-up and re-routed over many routing iterations.

[8]This derives from the original VPR formulation [1], which used a single (uniform) base resource cost for all wire lengths.

TABLE III: Normalized Impact of Exploration Limit on VTR benchmarks ($> 10K$ primitives, Normalized Geomean)

	W_{min}	Route Time (find W_{min})	Routed WL ($1.3 \cdot W_{min}$)	Crit. Path Delay ($1.3 \cdot W_{min}$)	Route Time ($1.3 \cdot W_{min}$)
static	1.00	1.00	1.00	1.00	1.00
dynamic	0.98	1.11	1.00	1.00	0.98

To address this issue we scaled the wire base costs to be proportional to each wire's length. The resulting congestion cost estimates for long wires produced by the map lookahead are shown in Figure 6c. These costs make long wires more expensive than short wires, particularly when used to travel distances shorter than their length.[9] This guides short distance and non-timing-critical connections to use the more plentiful short wires.

The impact of using length-scaled base costs on the two lookaheads is shown in the last two rows of Table II. With length-scaled base costs the map lookahead (`map_length`) significantly improves route-time, so it is 11% *faster* than the original classic lookahead. The classic lookahead's run-time (`classic_length`) is also improved, but the relative improvement is smaller. For both lookaheads the length-scaled base costs reduce routed wirelength by 7-8%.[10]

C. Adapting to Congestion

To reduce run-time the VPR router has historically restricted the search space for each connection to a fixed bounding box region derived from net's driver and sink locations. However this can make it difficult to resolve congestion, since it forms a hard limit which may prevent a connection from avoiding congestion, as shown in the left of Figure 7. To avoid this, AIR dynamically expands the search limit when a net uses a routing resource adjacent to a bounding box edge as shown in the right of Figure 7. This ensures no hard limit restricts how far nets can move out of the way to alleviate congestion.

Table III shows dynamic bounding boxes improve minimum routable channel width (W_{min}) by 2% on the VTR benchmarks [15] for a moderate increase in minimum channel width search time, and slightly reduces run-time in the less congested fixed channel width case.

[9]With the previous uniform base costs using an L16 wire to move 8 units was half the cost of using 2 L4 wires. Using length-scaled base costs the L16 wire is twice as expensive as using 2 L4s, which is intuitively consistent as half the L16 wire would be unused.

[10]This indicates VPR 7 sub-optimally allows short distance connections to use long wires. This was also independently identified and fixed by [10].

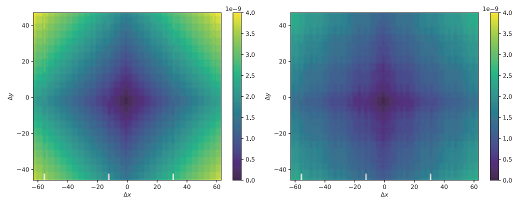

(a) Classic L4 (b) Map L4

Fig. 5: Lookahead delay estimates from short wires.

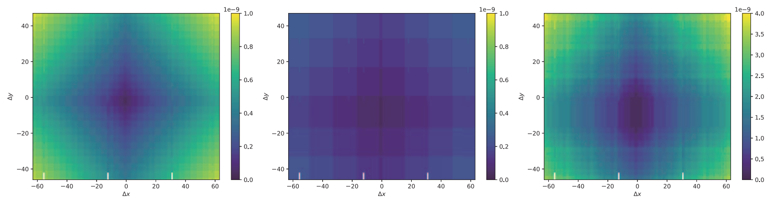

(a) Classic L4 (b) Classic L16 (c) Map Length L16

Fig. 6: Lookahead congestion estimates from short (L4) and long (L16) wires.

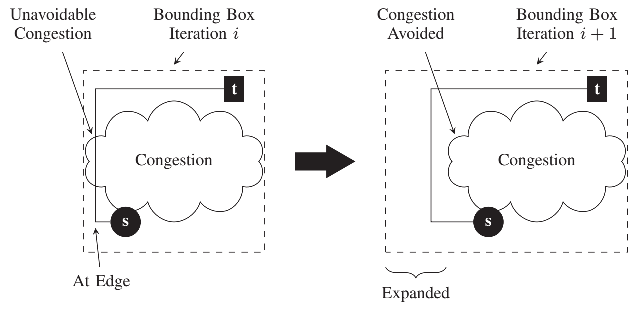


Fig. 7: Dynamic bounding box example.

IV. Multi-convergence Routing

During negotiated congestion routing, the router tries to resolve congestion while minimizing the impact on critical path delay. However in highly congested designs the critical path may be detoured to resolve congestion. To improve this behaviour AIR can perform multi-convergence routing. Instead of returning the first legal routing found, the router attempts to re-route critical connections to improve critical path delay. This may result in new congestion which could cause a huge amount of re-routing in a traditional negotiated congestion router. However AIR's incremental routing approach (Section II) keeps this well controlled.

When a legal routing is found (Algorithm 1 Line 8), the best routing is then updated[11] (Algorithm 1 Line 9). In preparation for the rip-up and re-routing of delay sub-optimal connections we adjust the costs of using routing resources by resetting the present congestion cost of currently-used resources to its low initial value (Algorithm 1 Line 13), and maintain the historical congestion costs which guide the router away from using previously congested resources. This allows timing critical connections to focus on finding low delay routes while still considering historically congested areas. Incremental re-routing then rips-up delay sub-optimal connections (Algorithm 1 Line 17). Once re-routing is 'kicked-off' in this manner any resulting congestion is naturally handled by incremental re-routing, and less critical (but newly congested) connections will be detoured away to resolve congestion. Finally, the best legal routing found is returned (Algorithm 1 Line 18).

Figure 8 shows the impact of multi-convergent routing on the large VTR benchmarks at various levels of routing stress. We can make several interesting observations.

Firstly, independent of multi-convergence routing, increasing channel width improves delay, wirelength, and run-time. The additional routing resources mean the router does not need to detour as drastically to resolve congestion.

[11]If it has lower critical path delay, with wirelength as a tie-breaker.

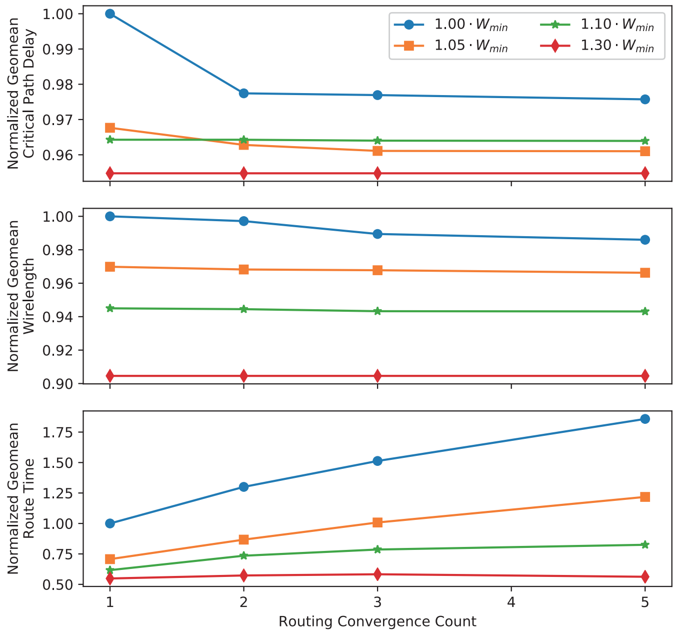


Fig. 8: QoR impact of multiple routing convergence on the large ($> 10K$ primitives) VTR benchmarks.

Secondly, multi-convergence routing improves critical path delay and wirelength in high stress settings at or near W_{min}. For instance, compared to a single convergence, allowing two convergences reduces critical path delay at W_{min} by 2.3%. However these gains diminish as channel width increases. Allowing more than two routing convergences offers minimal quality benefit.

Thirdly, multi-convergence routing is run-time efficient, with subsequent convergences increasing run-time by far less time than the initial convergence (which routed the entire netlist). For instance, at minimum channel width the second convergence only increased overall route-time by 30%. The run-time overhead of multi-convergence routing also decreases in less stressful routing conditions (where it offers less benefit). AIR's lazy routing optimizations (Section II) greatly reduce the work performed when a routing is almost legal, keeping the run-time overhead of multi-convergent routing low.

It is interesting to note that multi-convergence routing with delay-based rip-up accomplishes many of the same goals as the delay-targeted routing approach of [22]. In particular, it reduces the often chaotic impact of routing congestion on critical path delay at narrow channel widths. AIR's multi-convergence routing should be more run-time efficient as it lazily re-routes only the relevant connections in the netlist. In contrast delay-targeted routing re-routes the full netlist from scratch multiple times in search of an appropriate delay target. Furthermore, multi-convergence routing naturally extends to

TABLE IV: AIR Quality & Performance (Relative to VPR 7+)

Benchmark	Netlist Primitives	Clocks	Routed Wirelength		Routed CPD (clock geomean)		Route Time			
`gaussianblur`	1,860,332	1							■	△
`bitcoin_miner`	1,062,214	2	11,625,559	(0.87×)	0.96	(0.71×)	25.4	(0.23×)		
`directrf`	934,809	2							▲	△
`sparcT1_chip2`	766,899	1	7,331,842	(0.87×)	19.00	(0.94×)	8.5	(0.18×)		
`LU_Network`	630,446	19	5,994,360		1.26		53.6	(0.11×)		□
`LU230`	568,406	2	17,887,915	(0.90×)	9.89	(0.61×)	19.1	(0.06×)		
`mes_noc`	549,051	9	5,130,647	(0.84×)	9.74	(0.91×)	10.4	(0.08×)		
`gsm_switch`	491,990	4	6,451,664	(0.87×)	5.20	(0.71×)	5.7	(0.12×)		
`denoise`	344,210	1	3,784,910	(0.87×)	865.37	(0.92×)	5.8	(0.15×)		
`sparcT2_core`	288,477	1	4,547,421	(0.88×)	18.81	(1.00×)	4.5	(0.16×)		
`cholesky_bdti`	256,234	1	2,562,460	(0.85×)	9.73	(0.74×)	5.5	(0.15×)		
`minres`	252,736	2	2,703,947	(0.81×)	5.97	(0.81×)	3.3	(0.11×)		
`stap_qrd`	237,356	1	2,568,845	(0.86×)	7.70	(0.75×)	4.2	(0.19×)		
`openCV`	212,900	1	3,244,912	(0.83×)	10.13	(0.62×)	7.0	(0.16×)		
`dart`	202,481	1	2,390,767	(0.85×)	13.43	(0.88×)	3.0	(0.16×)		
`bitonic_mesh`	191,815	1	3,871,068	(0.80×)	13.68	(0.86×)	5.9	(0.14×)		
`segmentation`	168,637	1	1,985,817	(0.87×)	859.82	(0.92×)	3.1	(0.15×)		
`SLAM_spheric`	125,687	1	2,117,661	(0.86×)	86.04	(0.94×)	3.3	(0.13×)		
`des90`	109,960	1	2,079,998	(0.81×)	12.11	(0.83×)	3.4	(0.18×)		
`cholesky_mc`	108,575	1	1,219,835	(0.83×)	7.69	(0.82×)	2.4	(0.19×)		
`stereo_vision`	93,205	3	664,153	(0.84×)	3.28	(0.91×)	0.8	(0.14×)		
`sparcT1_core`	91,592	1	1,328,912	(0.89×)	8.54	(0.77×)	1.9	(0.15×)		
`neuron`	90,858	1	854,946	(0.83×)	6.12	(0.90×)	1.4	(0.18×)		
GEOMEAN	286,994.4	1.6	3,110,836.5	(0.85×)	12.82	(0.82×)	5.0	(0.14×)		

Normalized values relative to VPR 7+ in brackets; Run-time in Minutes; WL in grid tiles; CPD in ns;
▲ AIR time-out ($>$ 48 hrs); ■ AIR unroute; △ VPR 7+ time-out ($>$ 48 hrs); □ VPR 7+ unroute;

TABLE V: AIR & Academic Router Comparison

	Routed Wirelength	Routed CPD (worst clock)	Route Time
VPR 7+	1.00	1.00	1.00
CRoute [10]	0.89	0.95	0.30
AIR	0.85	0.81	0.15

Normalized geomean of mutually routable benchmarks

multi-clock designs where there is no longer a single delay target.

V. Experimental Results

To evaluate AIR's effectiveness we compare it to the routers from VPR 7+ [3], CRoute [10], and the industrial Intel Quartus 18.0 router [16]. For a fair run-time comparison experiments used the same Intel Xeon Gold 6146 based system, and all algorithms were run serially. All tools are evaluated using the Titan FPGA benchmarks [3] and a Stratix IV-like architecture, which are representative of modern FPGA usage. To isolate the effect of the VPR 7+ and AIR routers, identical packings and placements[12] and routing architectures are used, while wirelength and Critical Path Delay (CPD) metrics are calculated by VPR 8 [23].

Table IV compares AIR and the VPR 7+ router. On average, AIR significantly improves wirelength and critical path delay (CPD) by 15% and 18% respectively compared to VPR 7+, while run-time is drastically reduced (7.1× faster).

Table V compares AIR and CRoute's improvements relative to VPR 7. Note that CRoute results are from [10] which uses a different CAD flow for packing and placement – making it impossible to perfectly isolate the impact of the routers. With that caveat, the results show AIR's circuit implementations operate 17% faster and use 5% less wirelength than CRoute's. This was achieved while also completing routing 2.0× faster, and routing 2 more benchmarks (21 vs 19) than CRoute.

Finally, following the methodology from [3], we can compare AIR with the industrial Intel Quartus 18.0 router. Quartus results use Quartus' packing and placement which are higher quality/more routable than those produced by VPR [3]; as a result a perfect quality comparison can not be made. Despite routing from a lower quality placement, Table VI shows AIR completes routing 4.3× faster than the Quartus router – further illustrating its scalability. While the circuits routed by AIR use 27% more wirelength and operate 23% slower, this is due to the lower quality placement being used. This can be confirmed

[12]Generated by VPR 8 at iso-Quartus place-time.

TABLE VI: AIR & Quartus Comparison

	Routed Wirelength	Routed CPD (clock geomean)	Route Time
Quartus 18.0	1.00	1.00	1.00
VPR8 Place + AIR	1.27	1.23	0.23

Normalized geomean of mutually routable benchmarks

TABLE VII: AIR Congestion Oblivious vs Congestion Free

	Routed Wirelength	Routed CPD (worst clock)
Congestion Oblivious (min. delay)	1.00	1.00
Congestion Free (legal)	0.86	1.00

Normalized geomean of routable benchmarks

by comparing the quality of the initial routing (congestion oblivious and routed for minimum delay), and final legal routing (congestion free) produced by AIR.[13] Table VII shows the final legal routings produced by AIR do not degrade critical path delay or wirelength. In fact, wirelength improves since non-critical connections are re-routed for wirelength.

VI. Conclusions

We have presented AIR, the Adaptive Incremental Router, which uses a variety of techniques to improve router run-time and quality. AIR is a lazy router which avoids unnecessary work by re-routing nets incrementally and using spatial information to select only the relevant portions of route trees when routing high fanout net connections. AIR also adapts to the routing problem it is solving by adjusting per net search limits for congestion and building a lookahead which captures the characteristics of the target FPGA architecture. These techniques make it feasible to efficiently perform multi-convergence routing which improves the quality of existing routings, particularly in the presence of significant congestion.

Compared to the VPR 7 router, AIR runs 7.1× faster while reducing wirelength by 15% and critical path delay by 18%. AIR also produces higher quality implementations than a recent academic router [10], while reducing run-time by 2.0×. Finally, compared to the industrial Quartus router, AIR completes routing 4.3× faster while maintaining or improving quality.

Given AIR's efficient run-times and high quality it is used as the default router for VPR 8, and hence available as open-source software [24].

Acknowledgements

This work was supported by the NSERC/Intel Industrial Research Chair in Programmable Silicon, Huawei, the Canadian Foundation for Innovation, as well as NSERC CGS-D and Ontario Graduate scholarships. We would also like to thank Oleg Petelin for useful discussions.

References

[1] V. Betz, J. Rose *et al.*, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[2] D. Lewis, D. Cashman *et al.*, "Architectural Enhancements in Stratix V^{TM}," in *Field Programmable Gate Arrays*, 2013, pp. 147–156.

[3] K. E. Murray, S. Whitty *et al.*, "Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, pp. 10:1–10:18, 2015.

[13]Congestion oblivious routing produces a critical path delay which is a near lower bound for the given placement.

[4] L. McMurchie and C. Ebeling, “PathFinder: A Negotiation-based Performance-driven Router for FPGAs,” in *Field Programmable Gate Arrays*, 1995, pp. 111–117.

[5] G. G. Lemieux and S. D. Brown, “A detailed routing algorithm for allocating wire segments in field-programmable gate arrays,” in *Proc. Physical Design Workshop*, 1993, pp. 215–226.

[6] M. J. Alexander and G. Robins, “New performance-driven fpga routing algorithms,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1505–1517, 1996.

[7] V. Betz and J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” in *Field-Programmable Logic and Appl.*, 1997, pp. 213–222.

[8] K. So, “Enforcing long-path timing closure for fpga routing with path searches on clamped lexicographic spirals,” in *Field Programmable Gate Arrays*, 2008, pp. 24–34.

[9] X. Chen, J. Zhu *et al.*, “Timing-driven routing of high fanout nets,” in *Field Programmable Logic and Appl.*, 2011, pp. 423–428.

[10] D. Vercruyce, E. Vansteenkiste *et al.*, “CRoute: A Fast High-quality Timing-driven Connection-based FPGA Router,” in *Field-Programmable Custom Comput. Mach.*, 2019.

[11] M. Stojilovi, “Parallel FPGA Routing: Survey and Challenges,” in *Field Programmable Logic and Appl.*, 2017, pp. 1–8.

[12] M. Gort and J. H. Anderson, “Accelerating FPGA Routing Through Parallelization and Engineering Enhancements,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 31, no. 1, pp. 61–74, 2012.

[13] M. Shen and G. Luo, “Corolla: GPU-Accelerated FPGA Routing Based on Subgraph Dynamic Expansion,” in *Field Programmable Gate Arrays*, 2017, pp. 105–114.

[14] C. H. Hoo and A. Kumar, “ParaDRo: A Parallel Deterministic Router Based on Spatial Partitioning and Scheduling,” in *Field-Programmable Gate Arrays*, 2018, pp. 67–76.

[15] J. Luu, J. Goeders *et al.*, “VTR 7.0: Next Generation Architecture and CAD System for FPGAs,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, 2014.

[16] *Quartus 18.0*, Intel corporation, 2019. [Online]. Available: https://www.intel.ca/content/www/ca/en/software/programmable/quartus-prime/overview.html

[17] J. S. Swartz, V. Betz *et al.*, “A Fast Routability-driven Router for FPGAs,” in *Int. Symp. on Field Programmable Gate Arrays*, 1998, pp. 140–149.

[18] O. Petelin and V. Betz, “The Speed of Diversity: Exploring Complex FPGA Routing Topologies for the Global Metal Layer,” in *Field Programmable Logic and Appl.*, 2016, pp. 1–10.

[19] M. Wainberg and V. Betz, “Robust Optimization of Multiple Timing Constraints,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 34, no. 12, pp. 1942–1953, Dec 2015.

[20] P. E. Hart, N. J. Nilsson *et al.*, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Trans. on Syst. Sci. and Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.

[21] D. Lewis, V. Betz *et al.*, “The Stratix^{TM} Routing and Logic Architecture,” in *Field Programmable Gate Arrays*, 2003, pp. 12–20.

[22] R. Y. Rubin and A. M. DeHon, “Timing-driven Pathfinder Pathology and Remediation: Quantifying and Reducing Delay Noise in VPR-pathfinder,” in *Field Programmable Gate Arrays*, 2011, pp. 173–176.

[23] K. E. Murray, O. Petelin *et al.*, “VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling,” *ACM Transactions on Reconfigurable Technology and Systems*, 2019, In revision.

[24] “Verilog to Routing Project,” verilogtorouting.org.