Guaranteed Yet Hard to Find: Uncovering FPGA Routing Convergence Paradox

Shashwat Shrivastava, Stefan Nikolić, Sun Tanaka,
Chirag Ravishankar, Dinesh Gaitonde, Mirjana Stojilović

FCCM 2025, Fayetteville, Arkansas, USA

Routing Convergence Failure

Many designs take long to route and still, in the end, fail

```
ERROR: [Route 35-3] Design is not routable as its global congestion level is 6

Time (s): cpu = 10:18:07; elapsed = 10:19:43. Memory (MB): peak = 15725.180;
gain = 2896.867; free physical = 793741; free virtual = 1259249
```

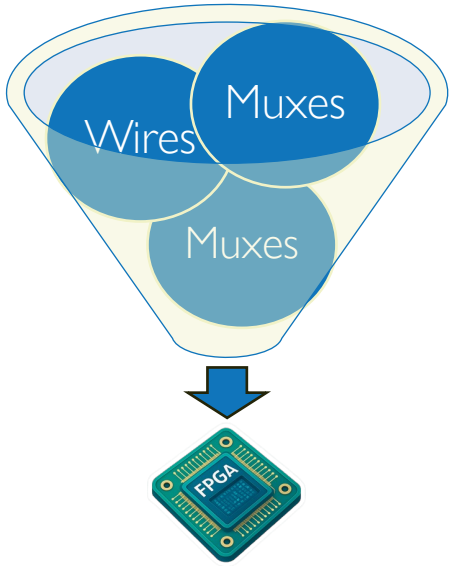
Is it due to the routing algorithm or poor routability of FPGA architecture?

Typical Convergence Fix

FPGA user level

Tuning parameters of the routing algorithm

FPGA company level



Adding routing resources or modifying the architecture

Little focus on finding fundamental inefficiencies in the routing algorithm

The Diagnosis Problem in Routing

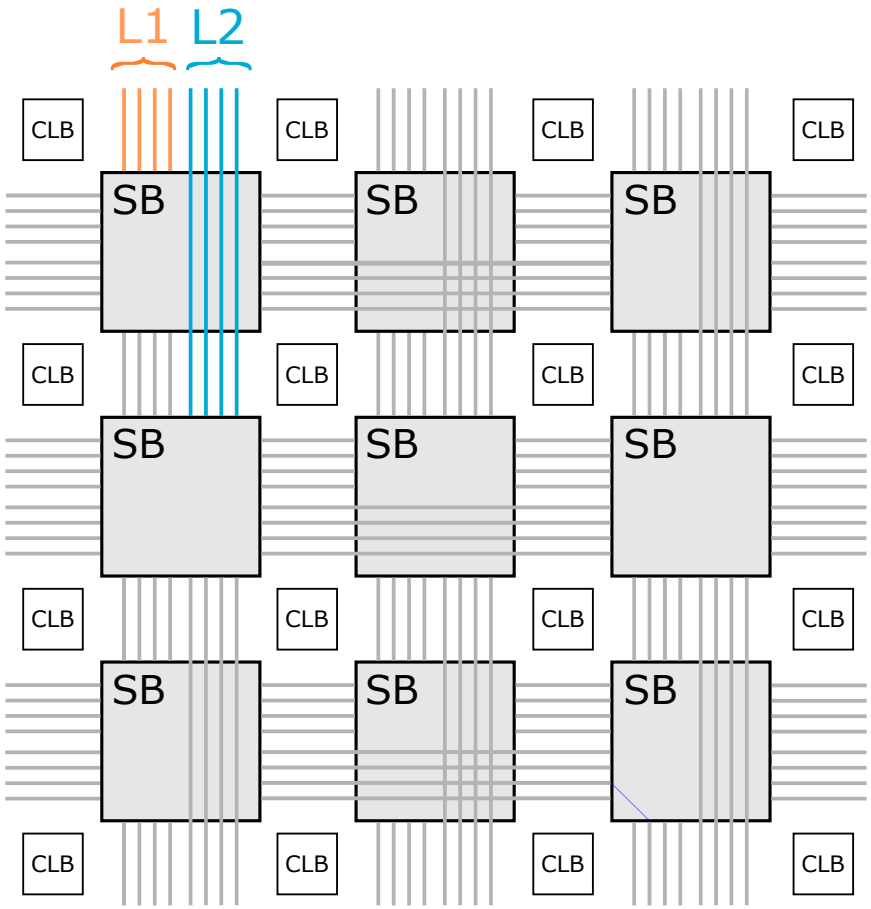
- No method to pinpoint the reason of convergence failure
- Lacking deeper understanding of the routing algorithm

> "Today, we are able to use the negotiated congestion widely and very successfully. Despite that, why it works so well eludes us at a theoretical level."
>
> -Sinan Kaptanoglu,
> FPGA and Reconfigurable Computing Hall-of-Fame Endorsement, 2012

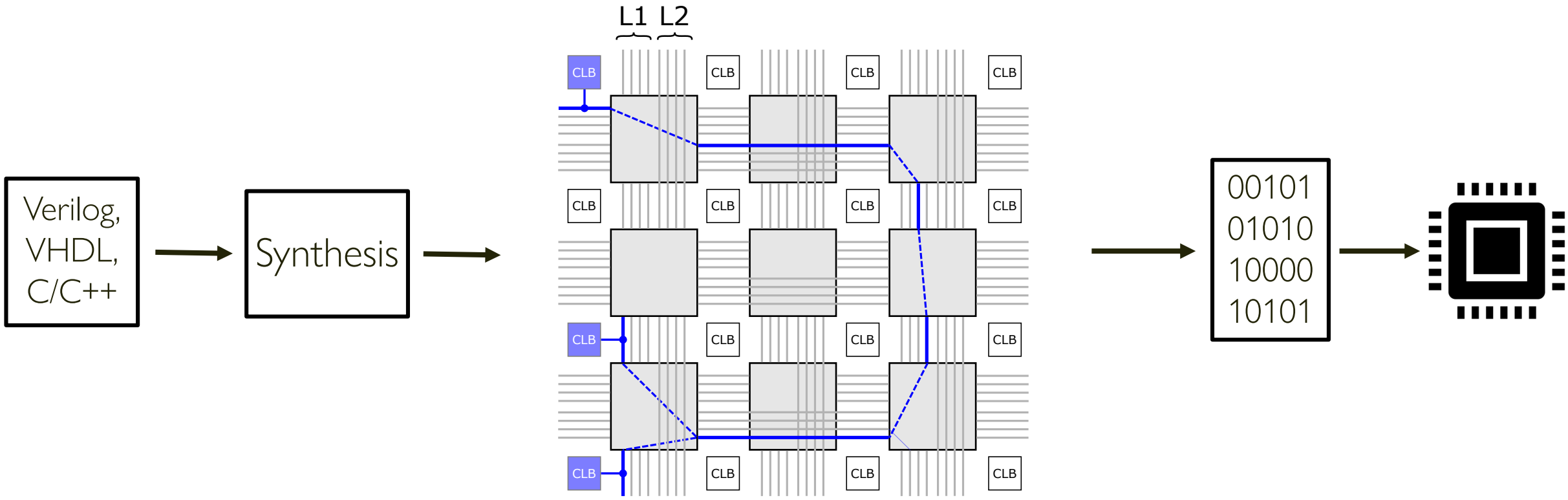
Journey to uncover inefficiencies lingering for >30 years

FPGA Architecture 101

- Configurable logic blocks (CLB)
  - LUTs, FFs
- Switch blocks (SB)
  - Routing multiplexers
- Wires of various length
  - 1, 2, 4, 12

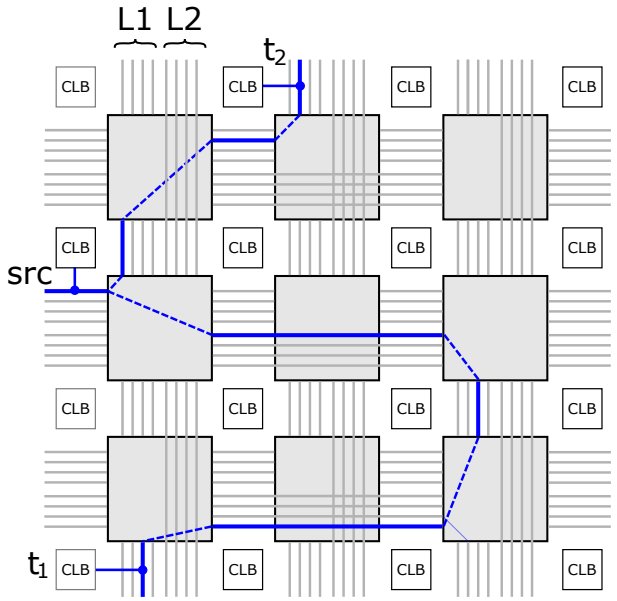


FPGA Compilation 101

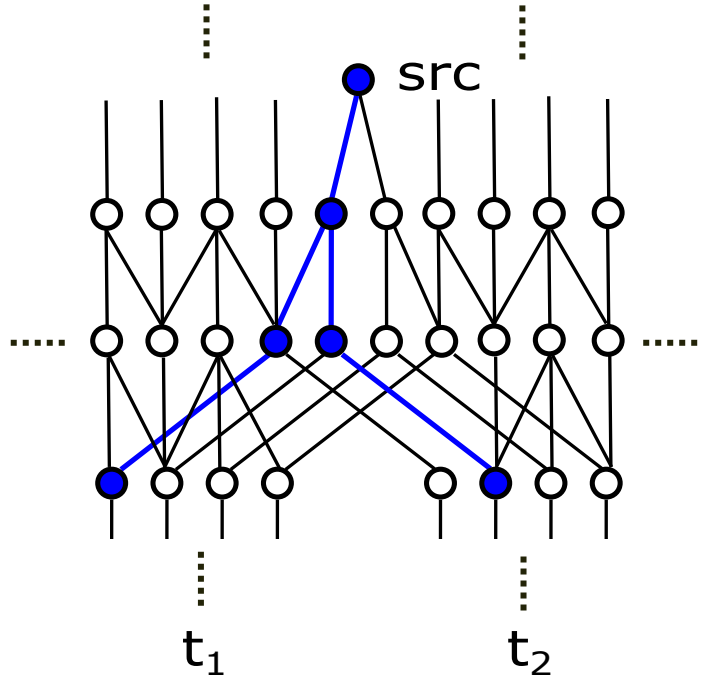


Routing 101

- Find approximate shortest disjoint paths between each net's source and its sinks



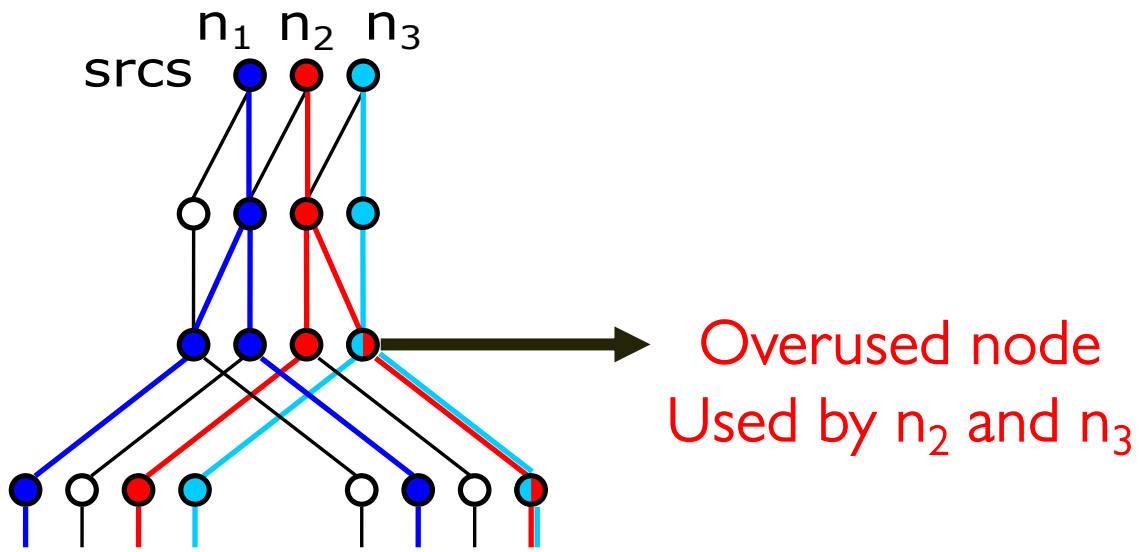
A net's routing tree



Routing graph

Routing 101 cont'd

- Find approximate shortest disjoint paths between each net's source and its sinks
- No nodes should be used by more than one net



- Algorithm → PathFinder [McMurchie and Ebeling, ISFPGA'95]

PathFinder: Algorithm

1
```
for iteration in max_iterations:
        for net in nets:
                route_tree = route_net(net)
                increase_cost_of_used_nodes()
        if no congestion:
                break
```

1 Route all nets iteratively

PathFinder: Algorithm

```
1 for iteration in max_iterations:
2       for net in nets:
              route_tree = route_net(net)
              increase_cost_of_used_nodes()
        if no congestion:
              break
```

1. Route all nets iteratively
2. Route nets sequentially

PathFinder: Algorithm

```
1 for iteration in max_iterations:
2       for net in nets:
              route_tree = route_net(net)
3             increase_cost_of_used_nodes()
      if no congestion:
            break
```

1. Route all nets iteratively
2. Route nets sequentially
3. Make used nodes expensive proportional to their overuse

PathFinder: Algorithm

```
1 for iteration in max_iterations:
2       for net in nets:
              route_tree = route_net(net)
3             increase_cost_of_used_nodes()
4       if no congestion:
              break
```

1. Route all nets iteratively
2. Route nets sequentially
3. Make used nodes expensive proportional to their overuse
4. Terminate, if no overused nodes → successful routing or

   max iterations reached → routing failure

PathFinder: Route Tree Construction

```
function route_net(net):
        route_tree = []
1       for sink in net:
                path = route_connection(route_tree, sink)
                route_tree = update_route_tree(path)
        return route_tree
```

1 Route each connection (representing a sink) sequentially

PathFinder: Route Tree Construction

```
function route_net(net):
        route_tree = []
1       for sink in net:
2               path = route_connection(route_tree, sink)
                route_tree = update_route_tree(path)
        return route_tree
```

1. Route each connection (representing a sink) sequentially
2. Connections use route tree constructed by previous connections

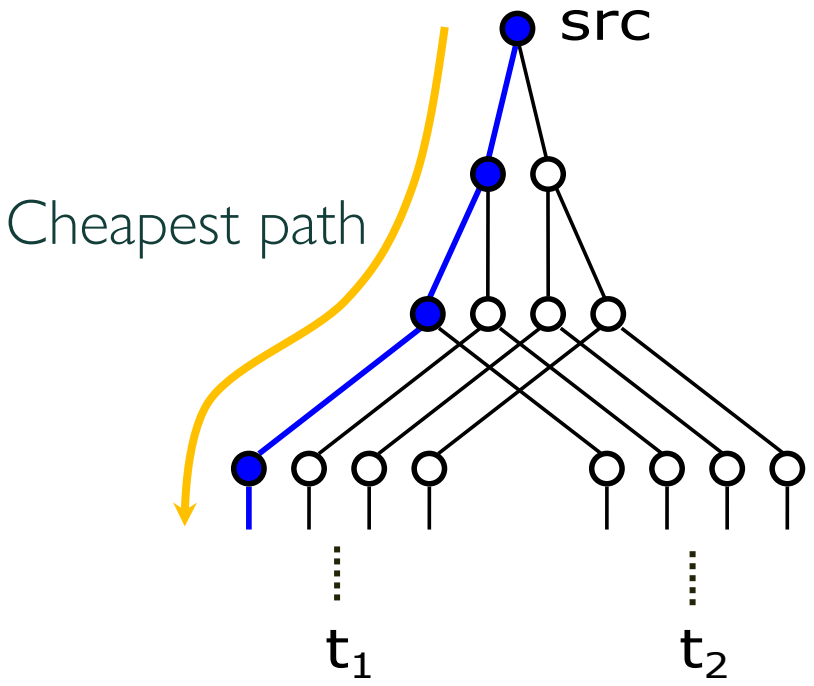
Example: Routing Multiple-Sink Net

- Assume a net with two sinks

- Order of routing sinks: t_1, t_2

Example: Routing Multiple-Sink Net

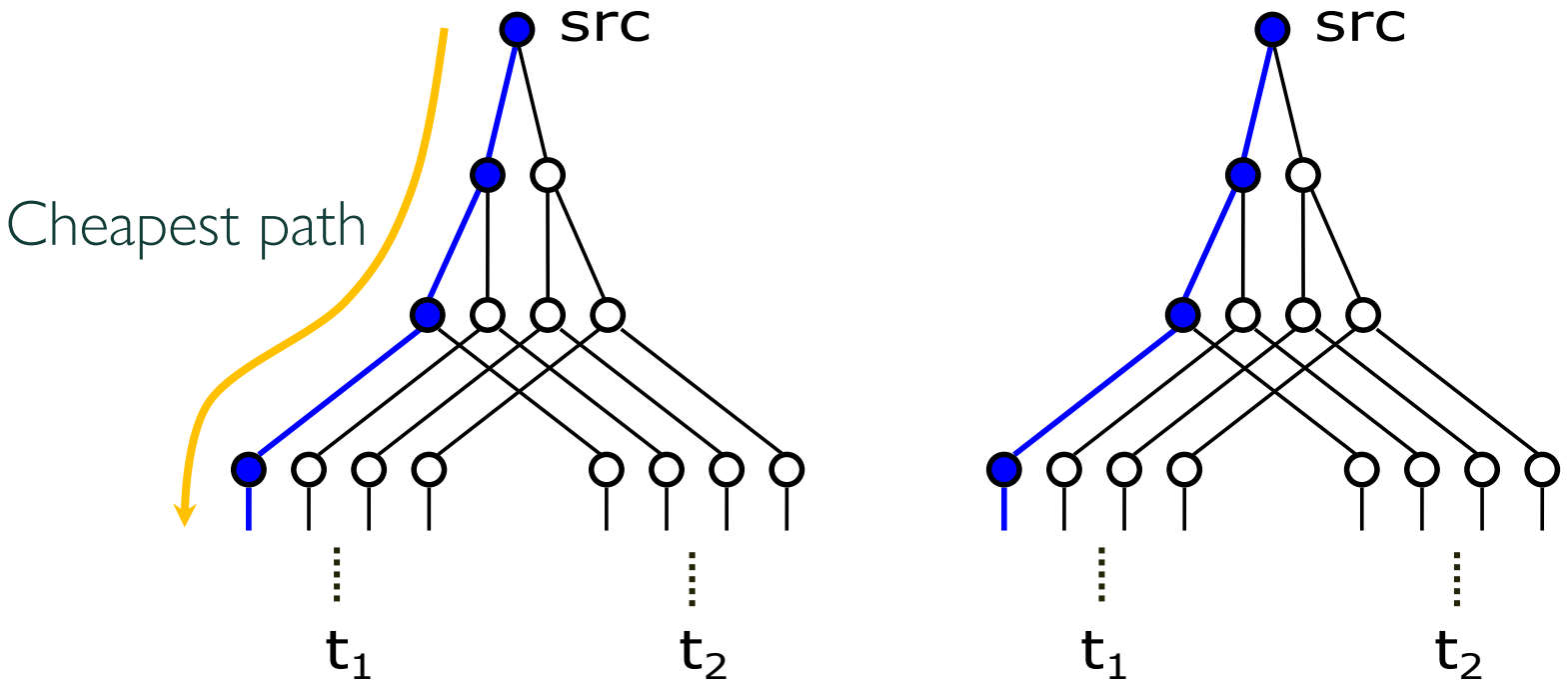
Order of routing sinks: t_1, t_2



Routing t_1 connection

Example: Routing Multiple-Sink Net

Order of routing sinks: t_1, t_2

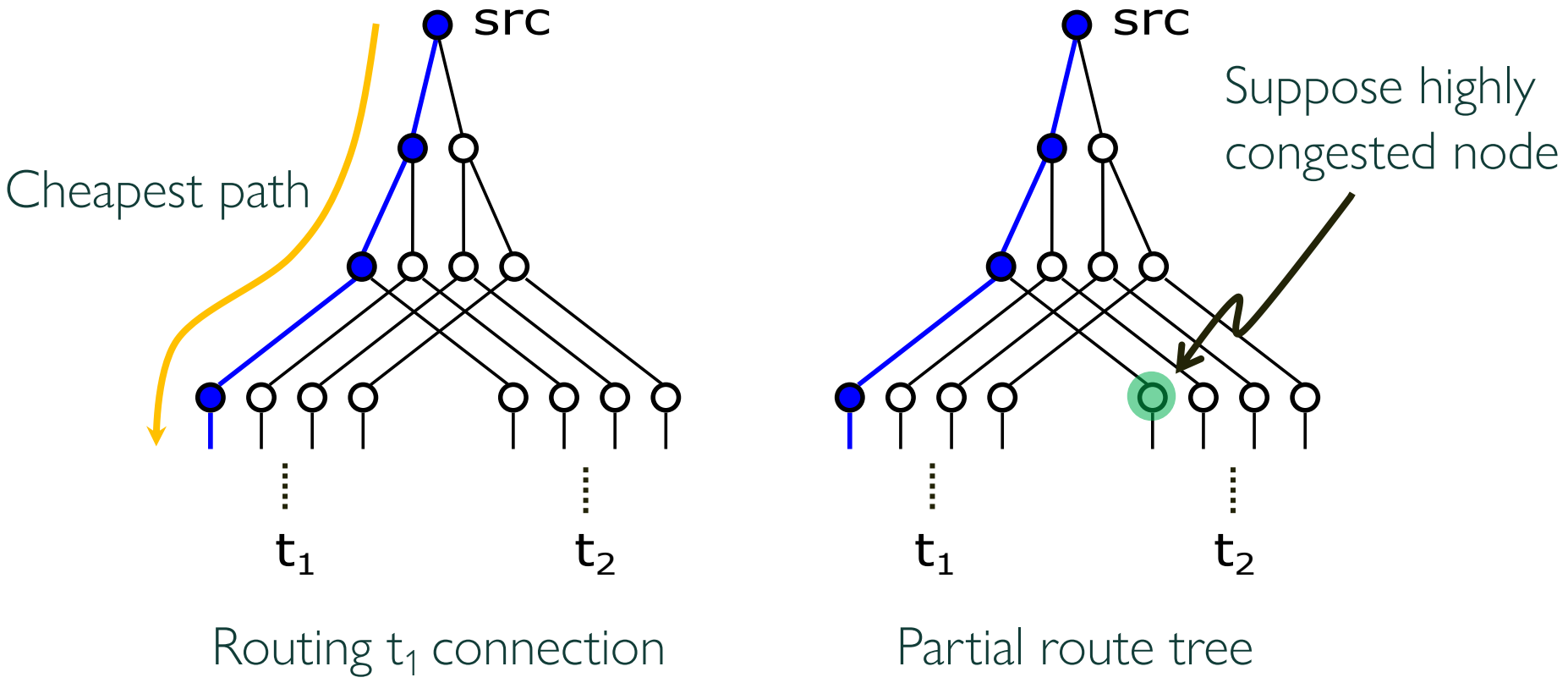


Routing t_1 connection

Partial route tree

Example: Routing Multiple-Sink Net

Order of routing sinks: t_1, t_2

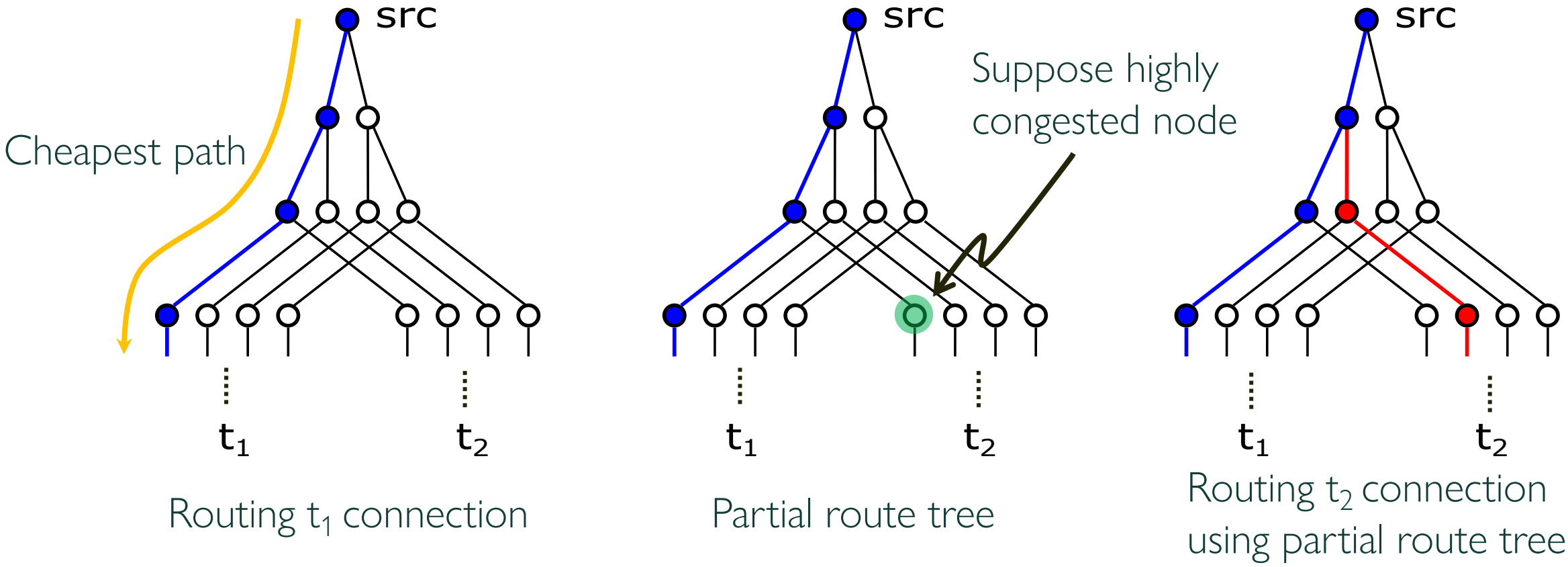


Routing t_1 connection

Partial route tree

Example: Routing Multiple-Sink Net

Order of routing sinks: t_1, t_2



Route trees are constructed greedily

We show that greedy route tree construction is inefficient for routing convergence

Diagnosing PathFinder Inefficiencies

- Develop a methodology to pinpoint the reason of convergence failure
  - Routing algorithm or FPGA routing architecture?

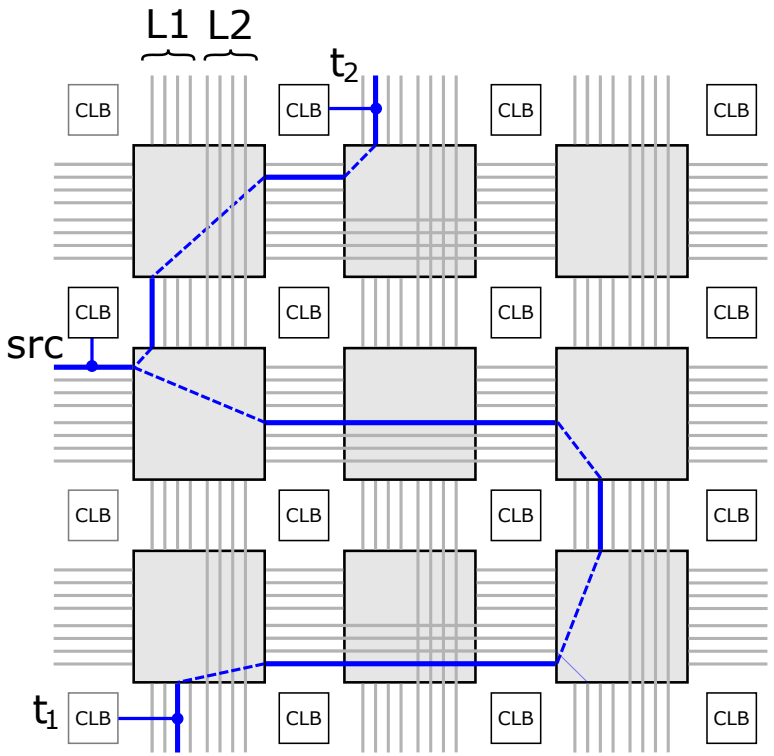
- Our solution
  - Eliminate FPGA routing architecture as a reason of convergence failure
    - How? Guarantee the existence of a routing solution for each design
  - Reduce the search space for deeper analysis
    - How? Create tractable routing problems amenable for manual inspection

Outline

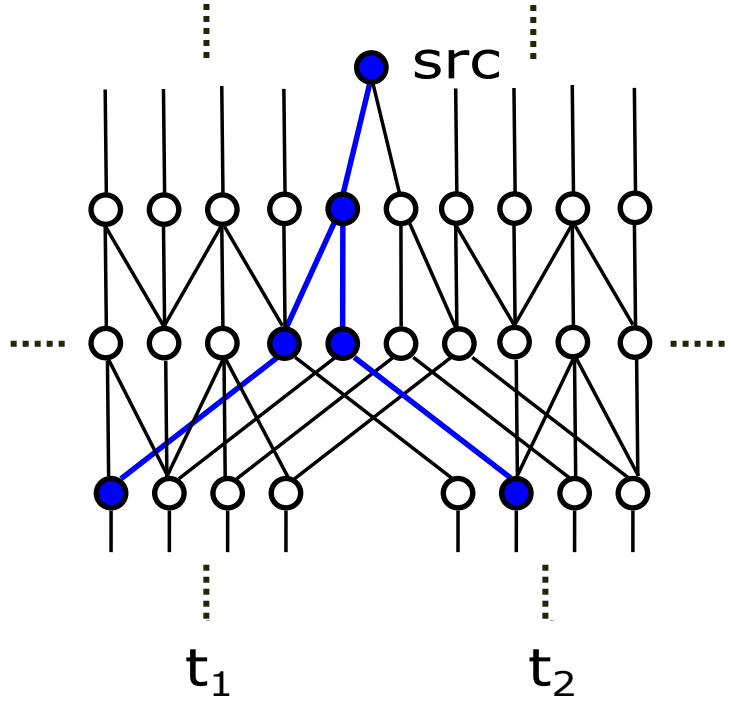
- Inefficient Route Trees
- Diagnostic Methodology
- Fixing Route Tree Construction
- Effect on Standard Setting

Diagnostic Methodology: Legal Path

1 Run routing in a standard setting to get congestion-free routing paths



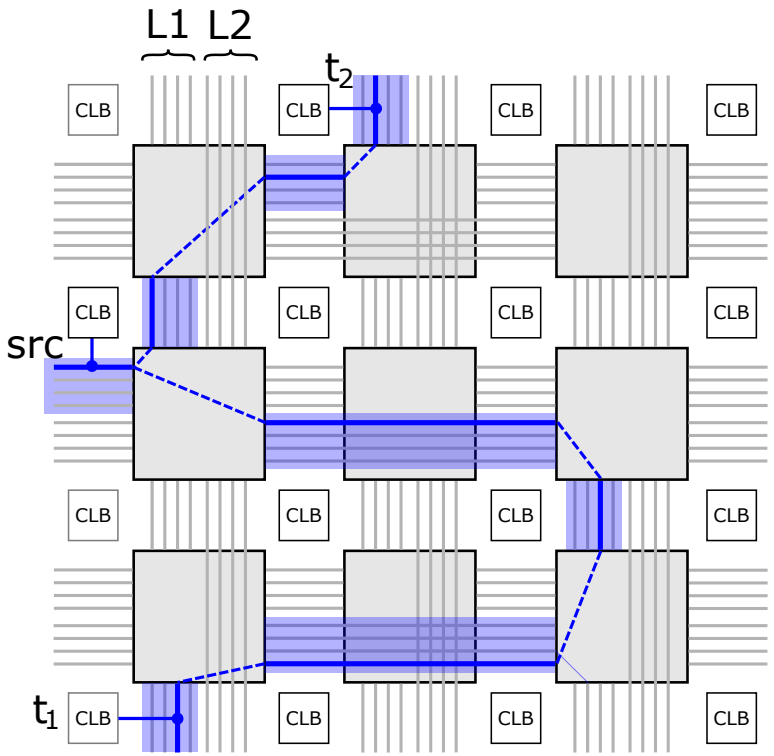
FPGA view



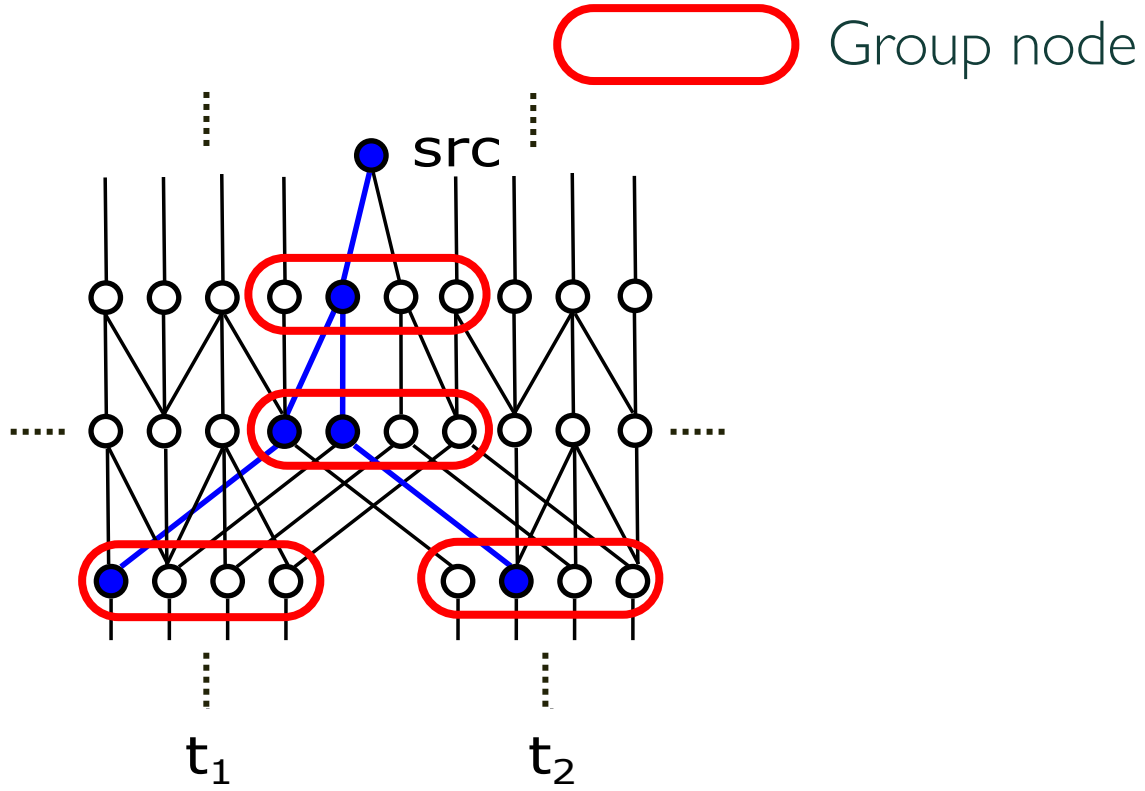
Graph view

Diagnostic Methodology: Grouping

2 Routing path is grouped with other wires to construct constrained regions



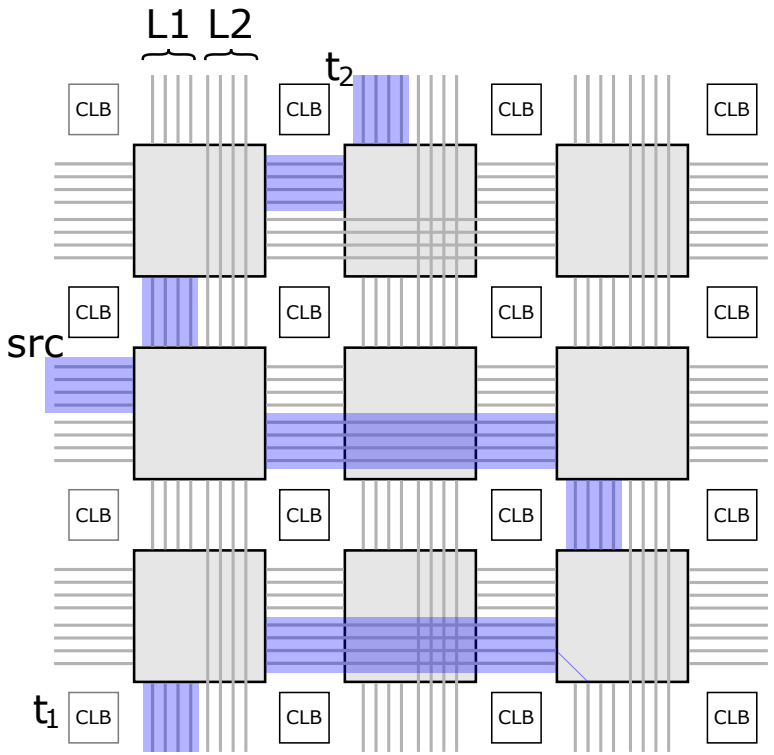
Wires of same length, same direction, originating from same SB are grouped



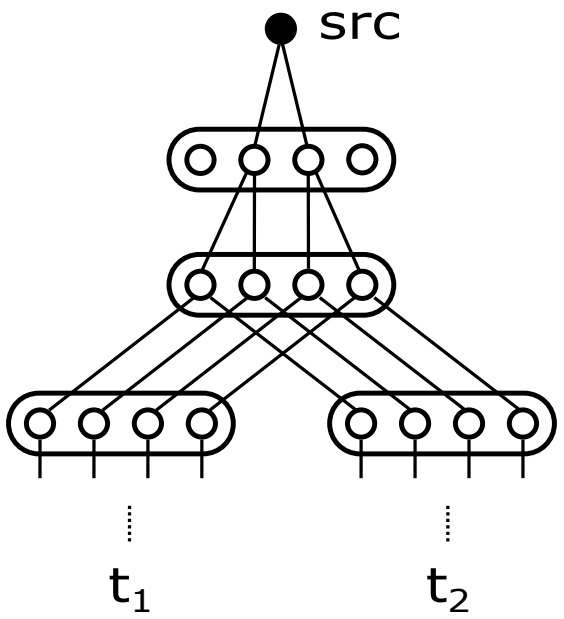
Group nodes with four routing nodes of the same type

Diagnostic Methodology: Constrained Routing

3 Router is instructed to route each connection only along the constrained routing region



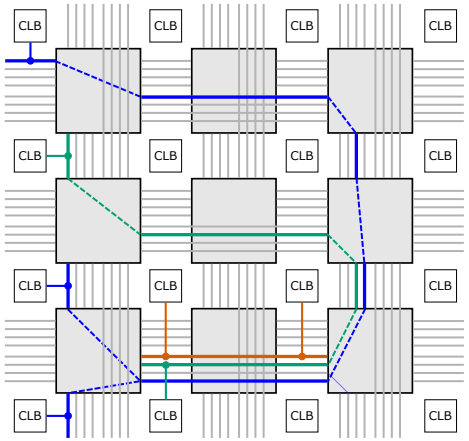
FPGA view

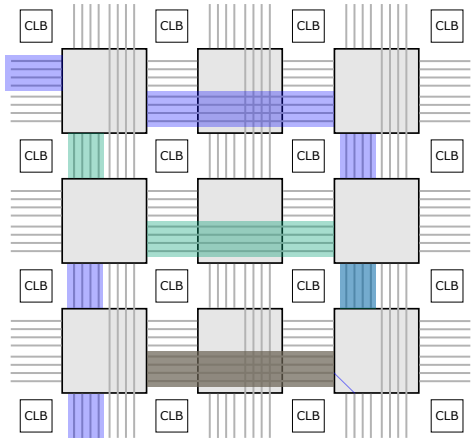


Graph view

Diagnostic Methodology: Overall

For each net, a constrained routing region is determined





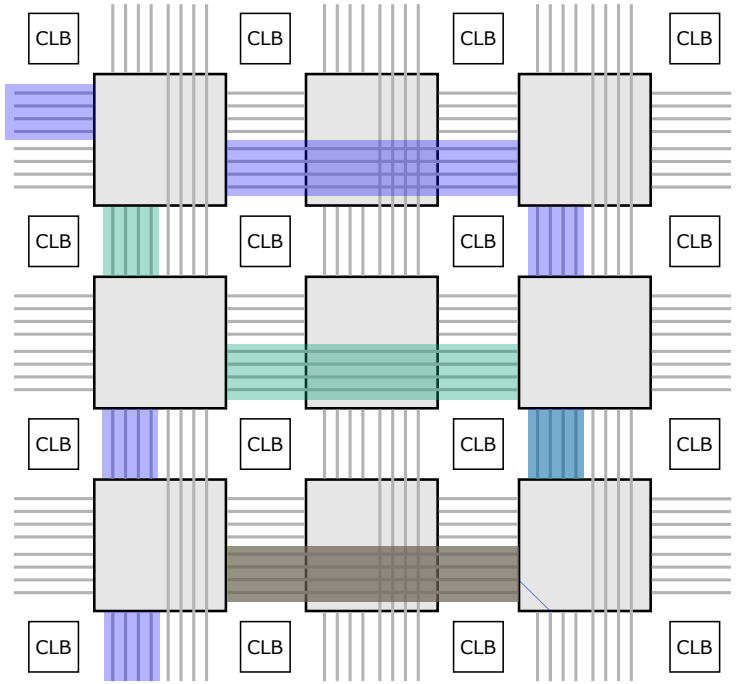
Routing in unconstrained region → Generate constrained regions → Routing in constrained region

- Netlist
- Placement

- Netlist
- Placement

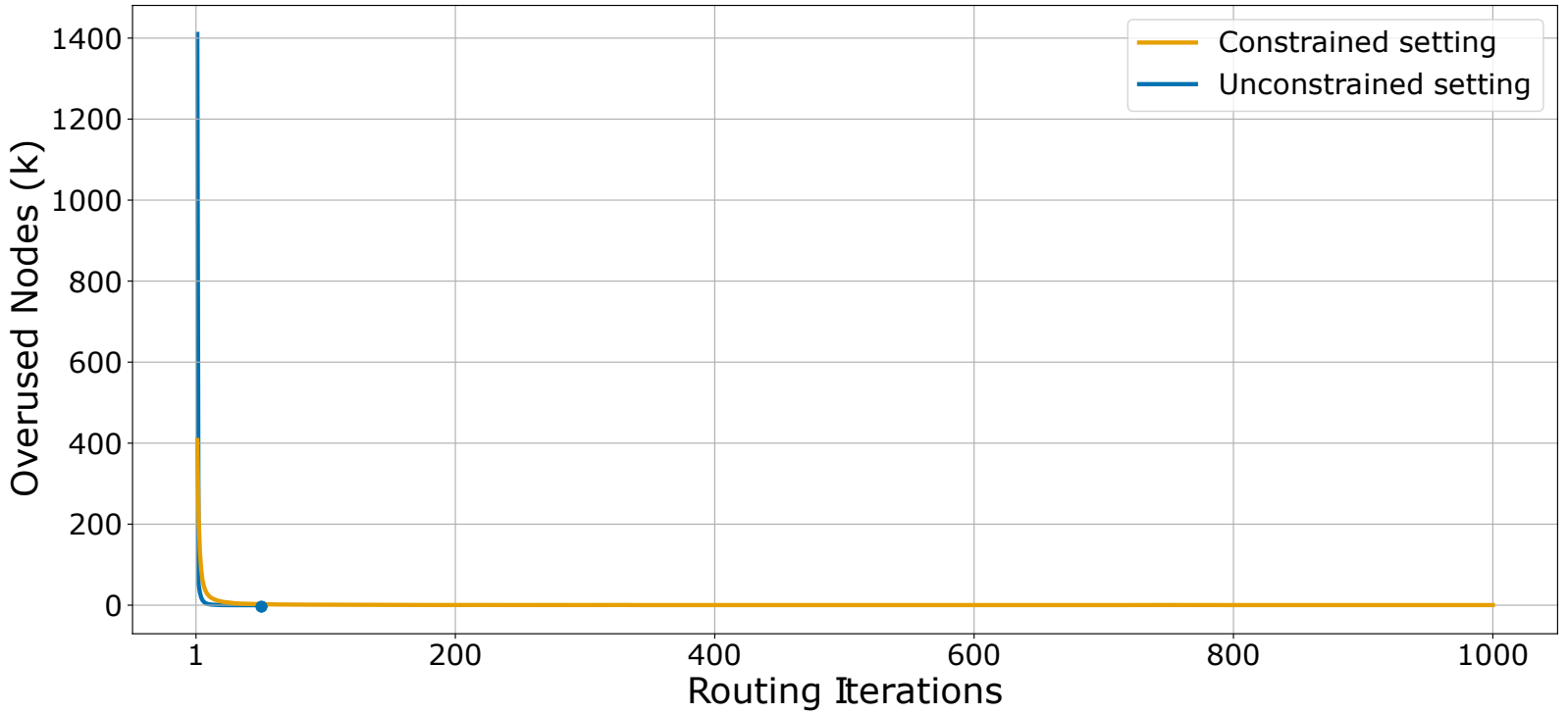
Routing in the Constrained Region: Expectation

- Routing regions small → reduced search space
  - Quick to route compared to unconstrained setting
- PathFinder is adaptive → should find the paths
- However, when we route …



Routing Convergence Paradox

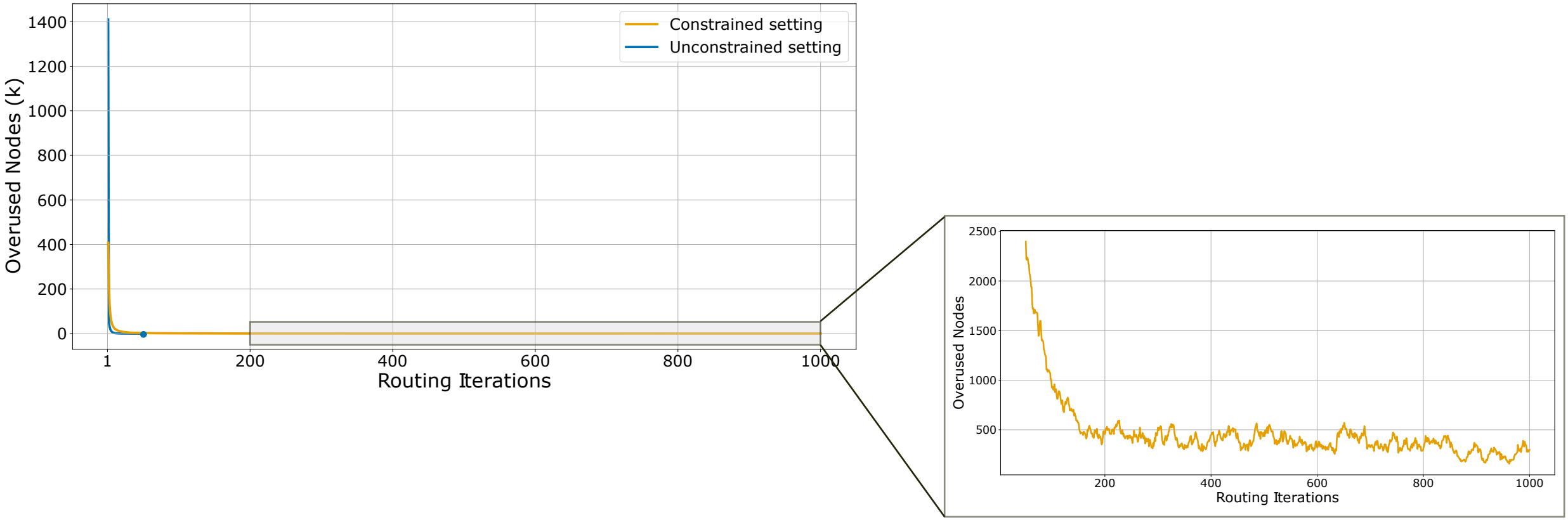
For one of the largest benchmarks from ISPD16 suite



11 out of 12 benchmarks do not converge

Routing Convergence Paradox

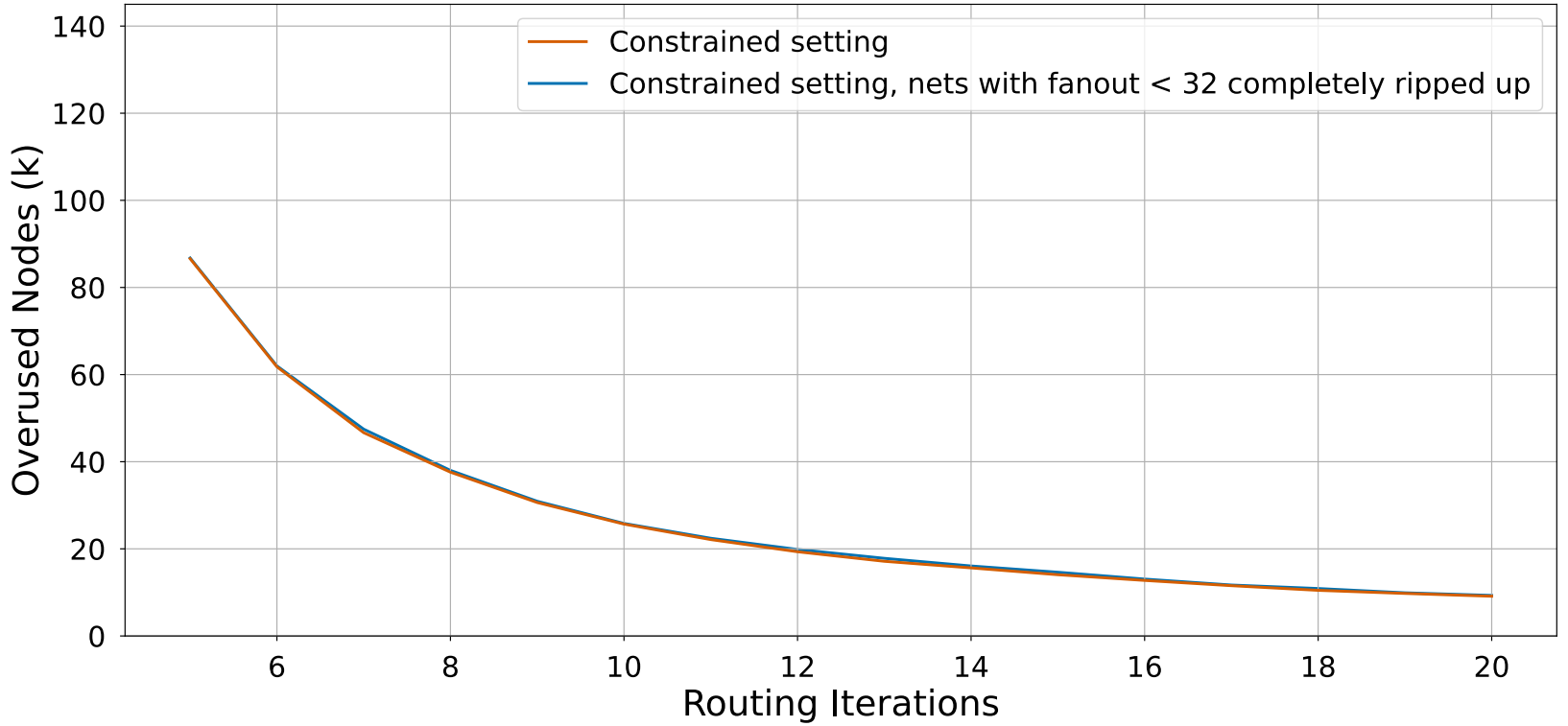
For one of the largest benchmarks from ISPD16 suite



Guaranteed yet hard to find!

PathFinder Tuning: Attempt #1

- Previous configuration (default): Rip up congested nets completely if fanout < 16
- New configuration: Rip up congested nets completely if fanout < 32

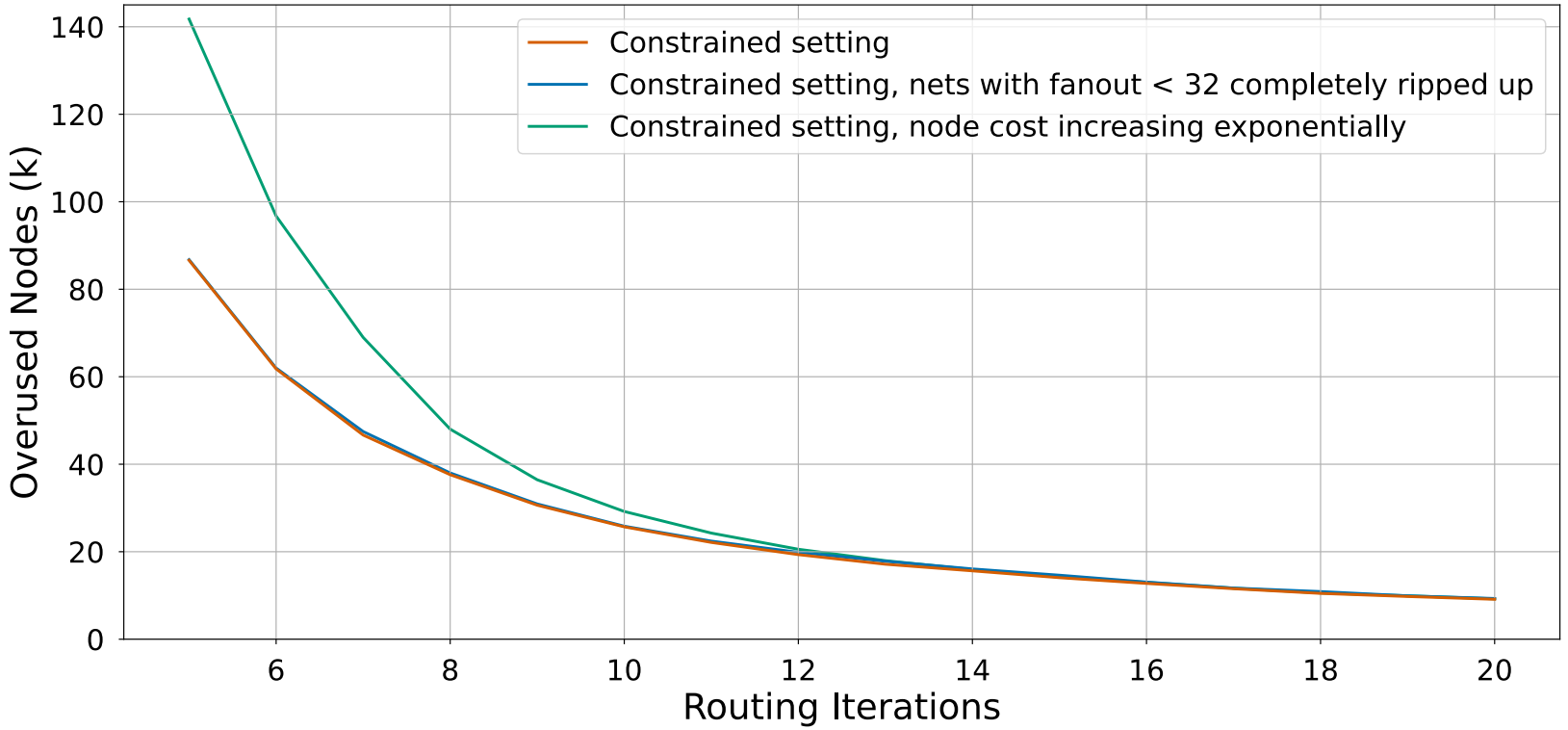


No new benchmarks converge

PathFinder Tuning: Attempt #2

Previous configuration: Fixed present congestion cost [Zha and Li, ISFPGA'22]

New configuration: Increase present congestion cost exponentially



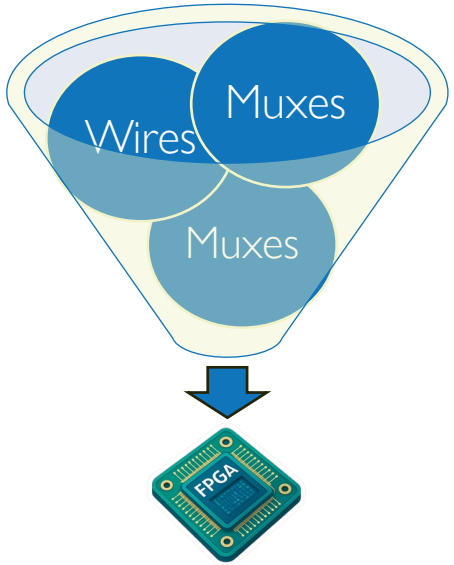
PathFinder tuning does not fix routing convergence problem

Where is the Problem?

PathFinder parameters ✗

Tuning does not help

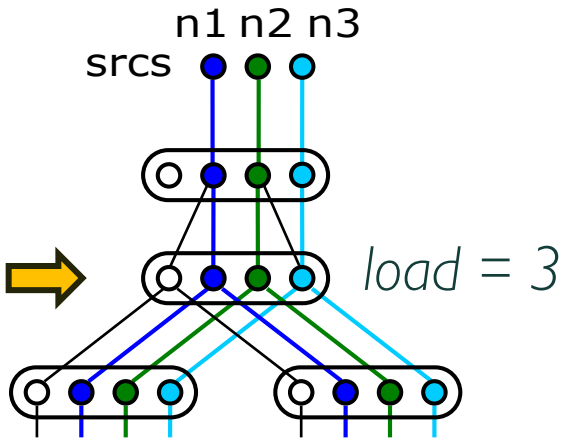
FPGA architecture ✗



Guaranteed legal routing solution
in the constrained region

Hinting at a fundamental inefficiency in PathFinder

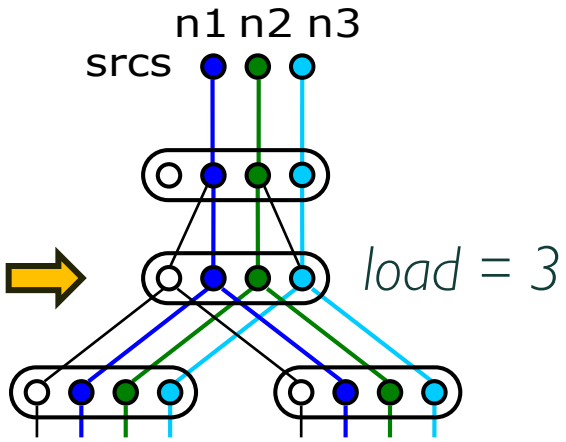
Closely Analyzing Routing Regions



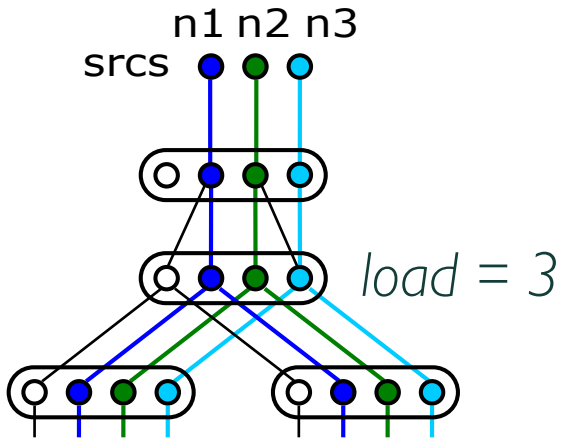
Legal paths of three nets before grouping

load: number of nodes used by at least one net within a grouped node

Closely Analyzing Routing Regions



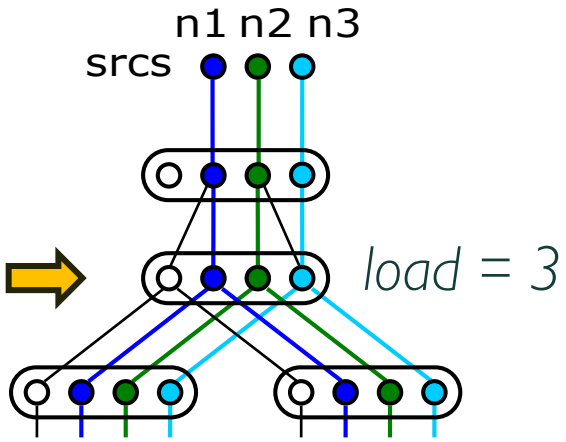
Legal paths of three nets before grouping



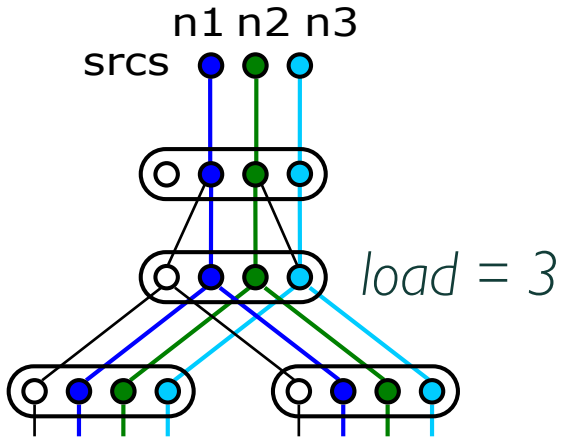
1 Initial load

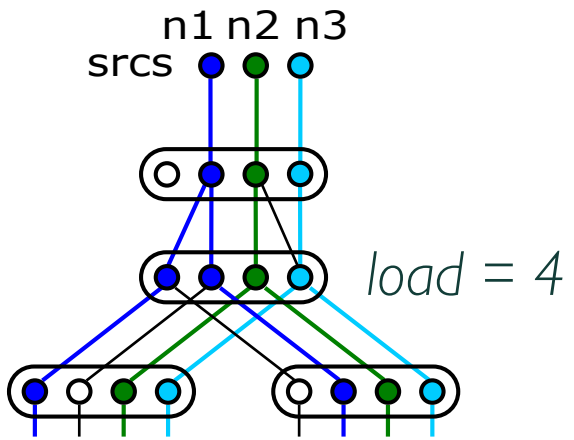
States of grouped nodes possible during routing in constrained regions

Closely Analyzing Routing Regions



Legal paths of three nets before grouping





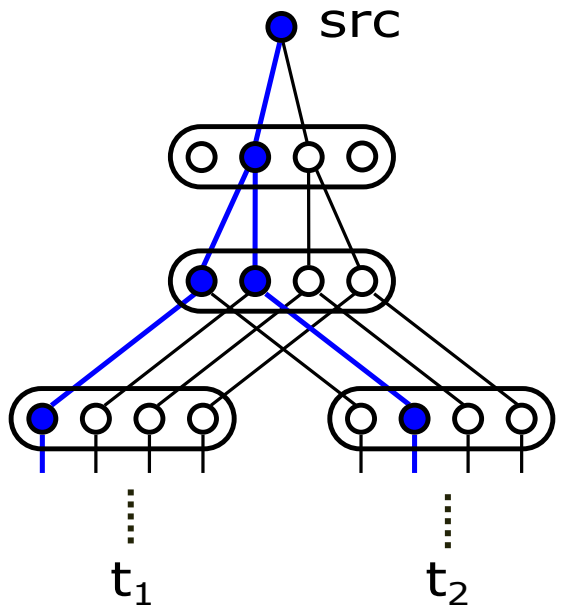
1 Initial load

2 Increased load

States of grouped nodes possible during routing in constrained regions

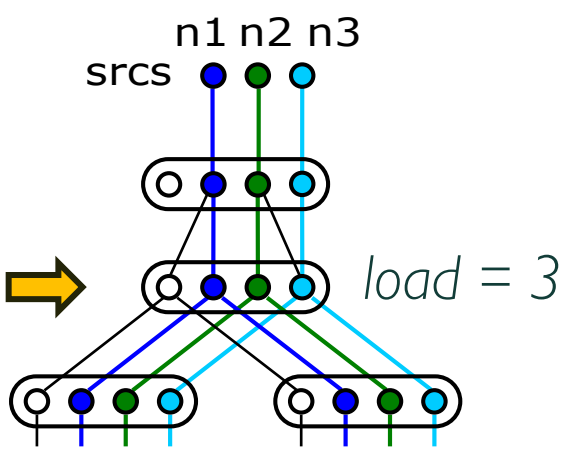
Load Increase

- PathFinder → greedy connection-based routing
- Connections of a net route oblivious of subsequent connections of the same net

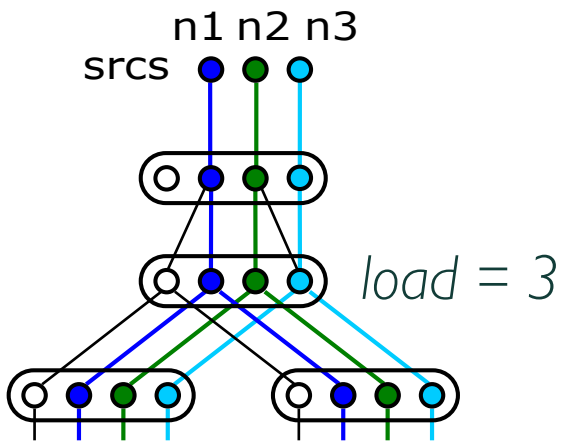


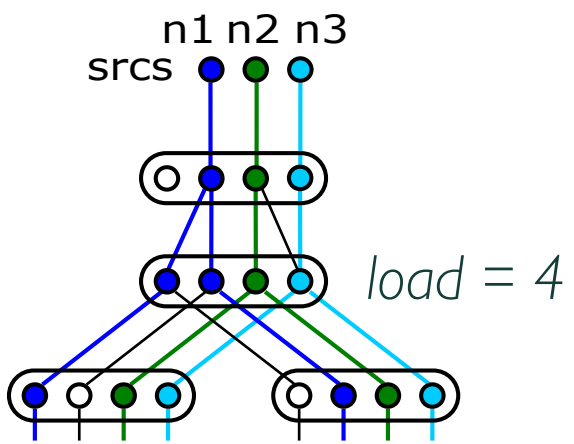
Greedy route tree construction results in load increase

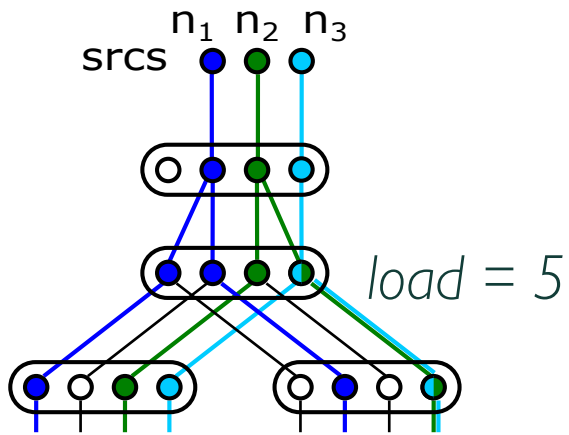
Closely Analyzing Routing Regions



Legal paths of three nets before grouping







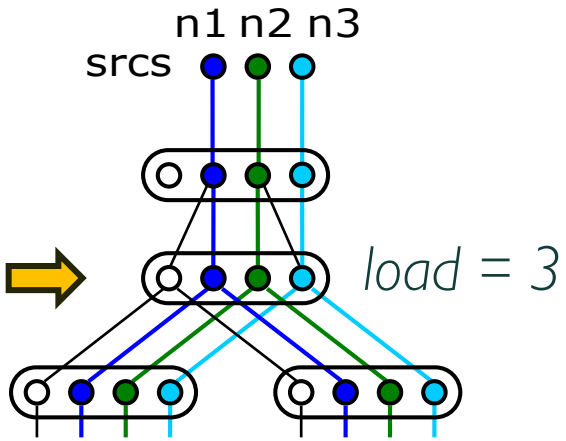
1 Initial load

2 Increased load

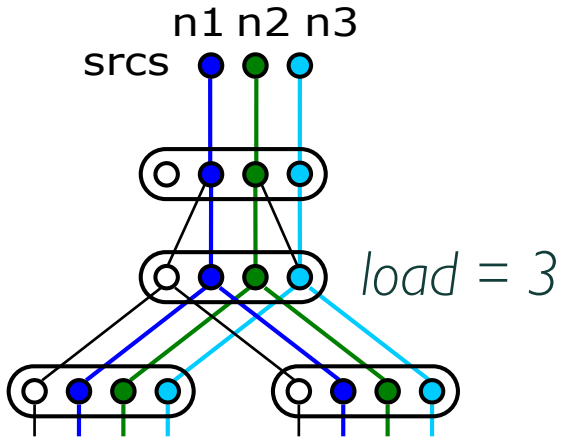
3 Overload: *load > capacity*

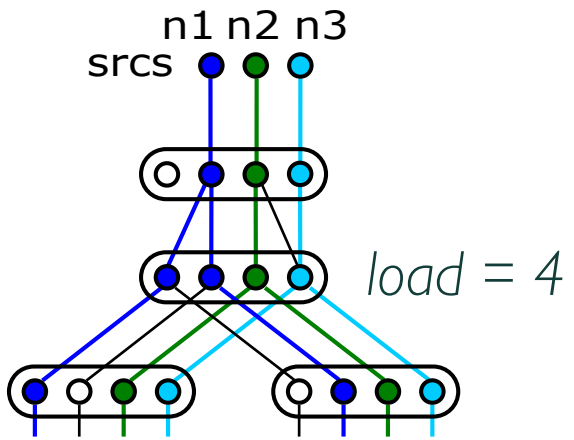
States of grouped nodes possible during routing in constrained regions

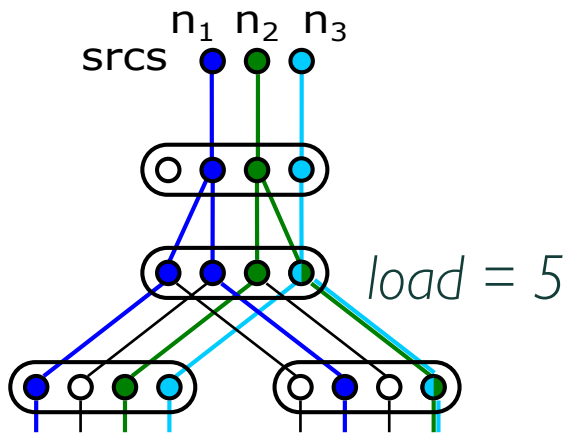
Closely Analyzing Routing Regions



Legal paths of three nets before grouping







1 Initial load

2 Increased load

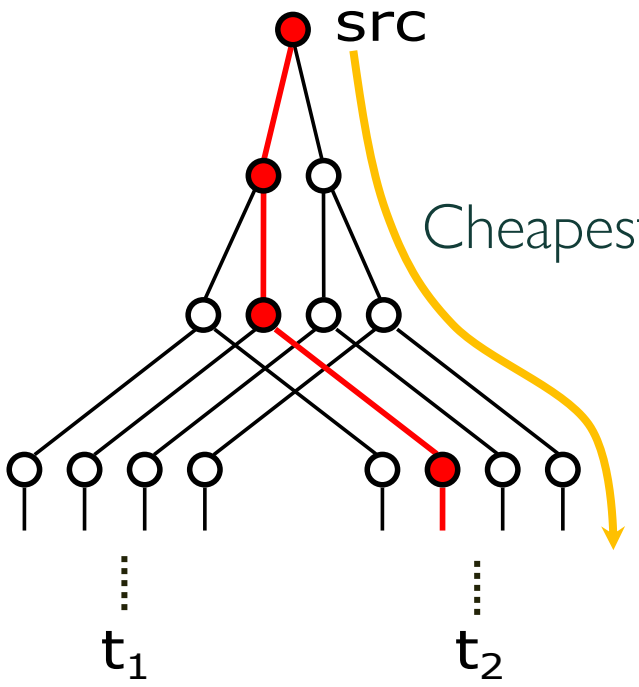
3 Overload: *load > capacity*

States of grouped nodes possible during routing in constrained regions

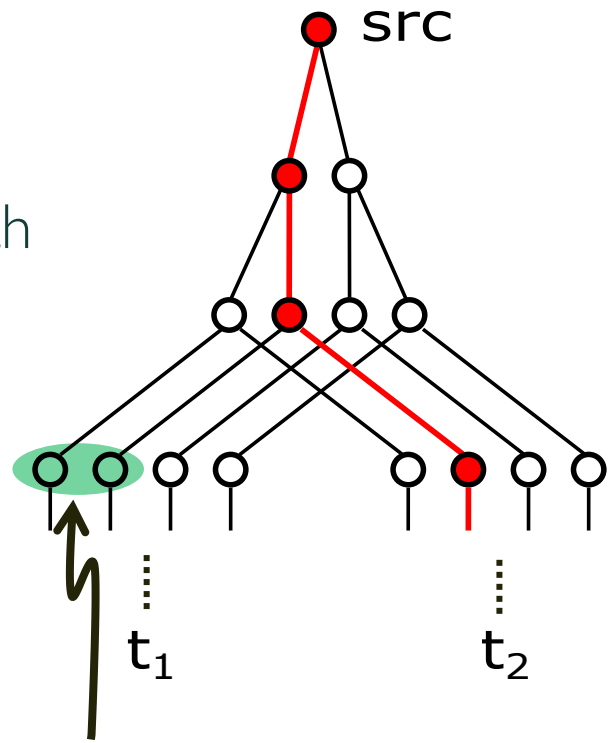
Load increase prevents routing from converging

Decreasing the Load

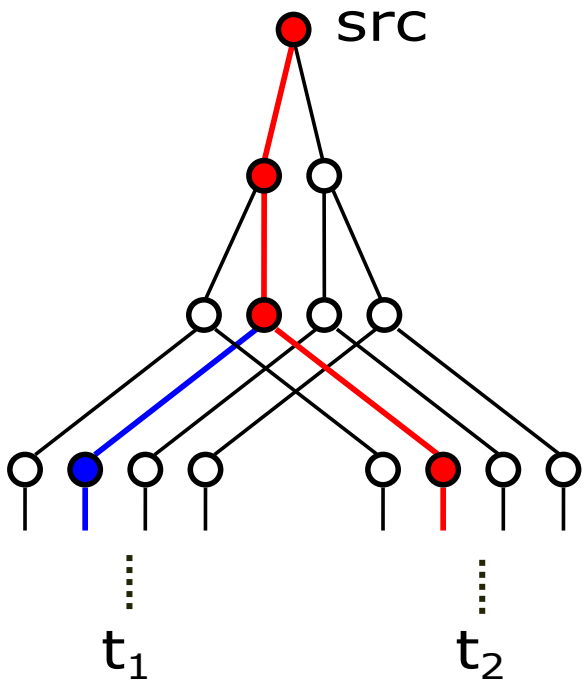
Change sink order from (t_1, t_2) to (t_2, t_1)



Routing t_2 connection

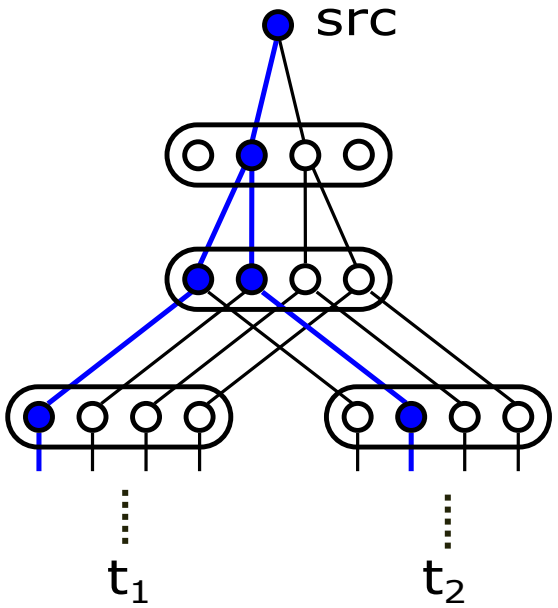


Suppose equivalent cost



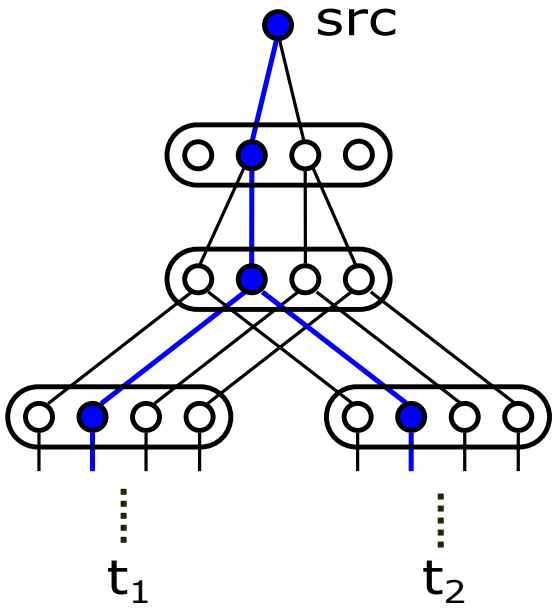
Routing t_1 connection using partial route tree

Efficient Route Tree



Sink order: t_1, t_2

Route tree with fewer nodes



Sink order: t_2, t_1

Changing sink orders can result in more efficient route trees

Outline

- Inefficient Route Trees
- Diagnostic Methodology
- Fixing Route Tree Construction
- Effect on Standard Setting

Route All Sink Permutations

- Ideally, for each net, route all permutations of sink orders
  - Each sink order can be routed in parallel
- But, super exponential growth in permutations for high-fanout nets
  - Not scalable
- Route only a subset of all possible sink orders
  - To generate a subset, randomly select multiple sink orders

Route Multiple Random Sink Shuffles

```
for iteration in max_iterations:
        for net in nets:
1               multiple_sink_orders = generate_random_shuffles(sinks)
                for sink_order in multiple_sink_orders:
                        route_tree = route_net(sink_order)
                        total_nodes, cong_cost = get_tree_cost(route_tree)
                        map[total_nodes, cong_cost] = sink_order
                route_tree = tree_selection(map)
```

1 Generate multiple random sink shuffles

$t_1, t_2, t_3, \ldots t_n$
$t_5, t_1, t_n, \ldots t_3$
.
.
.
$t_n, t_1, t_3, \ldots t_5$

Route Multiple Random Sink Shuffles

```
for iteration in max_iterations:
        for net in nets:
1               multiple_sink_orders = generate_random_shuffles(sinks)
2               for sink_order in multiple_sink_orders:
                        route_tree = route_net(sink_order)
                        total_nodes, cong_cost = get_tree_cost(route_tree)
                        map[total_nodes, cong_cost] = sink_order
                route_tree = tree_selection(map)
```

1. Generate multiple random sink shuffles
2. Route each random sink order

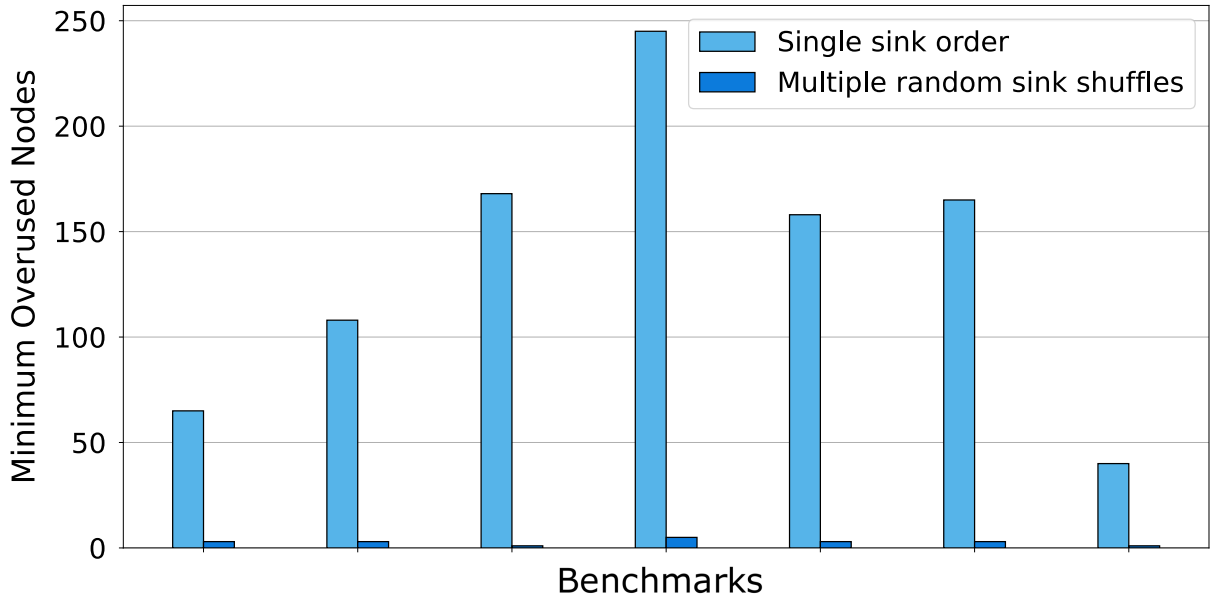
Route Multiple Random Sink Shuffles

```
for iteration in max_iterations:
        for net in nets:
1               multiple_sink_orders = generate_random_shuffles(sinks)
2               for sink_order in multiple_sink_orders:
                        route_tree = route_net(sink_order)
                        total_nodes, cong_cost = get_tree_cost(route_tree)
                        map[total_nodes, cong_cost] = sink_order
3               route_tree = tree_selection(map)
```

1. Generate multiple random sink shuffles
2. Route each sink order
3. Select the tree with minimum number of nodes

Results: Routing in Constrained Setting

- Five out of 12 benchmarks converged
- For the rest, the overused nodes dropped to single digits
  - Given the probabilistic nature of randomly sampling a sink order, if run with higher number of permutations they could route too



Constructing route trees with fewer nodes fixes convergence issues

Outline

- Inefficient Route Trees
- Diagnostic Methodology
- Fixing Route Tree Construction
- Effect on Standard Setting

Transfer to Standard Setting

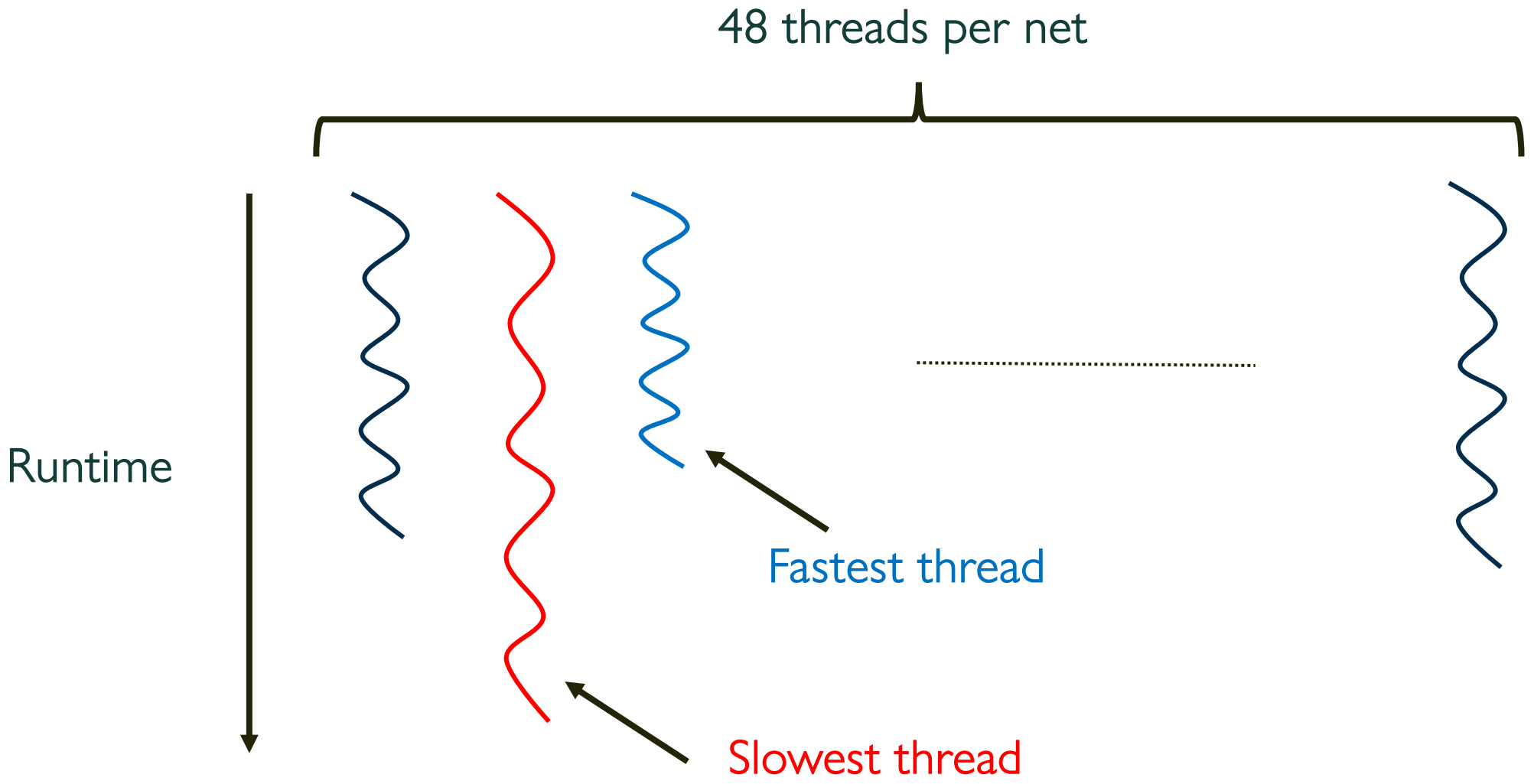
- Standard setting → unconstrained setting (no search space constraints)

- If route tree matters → should give improvements in standard setting

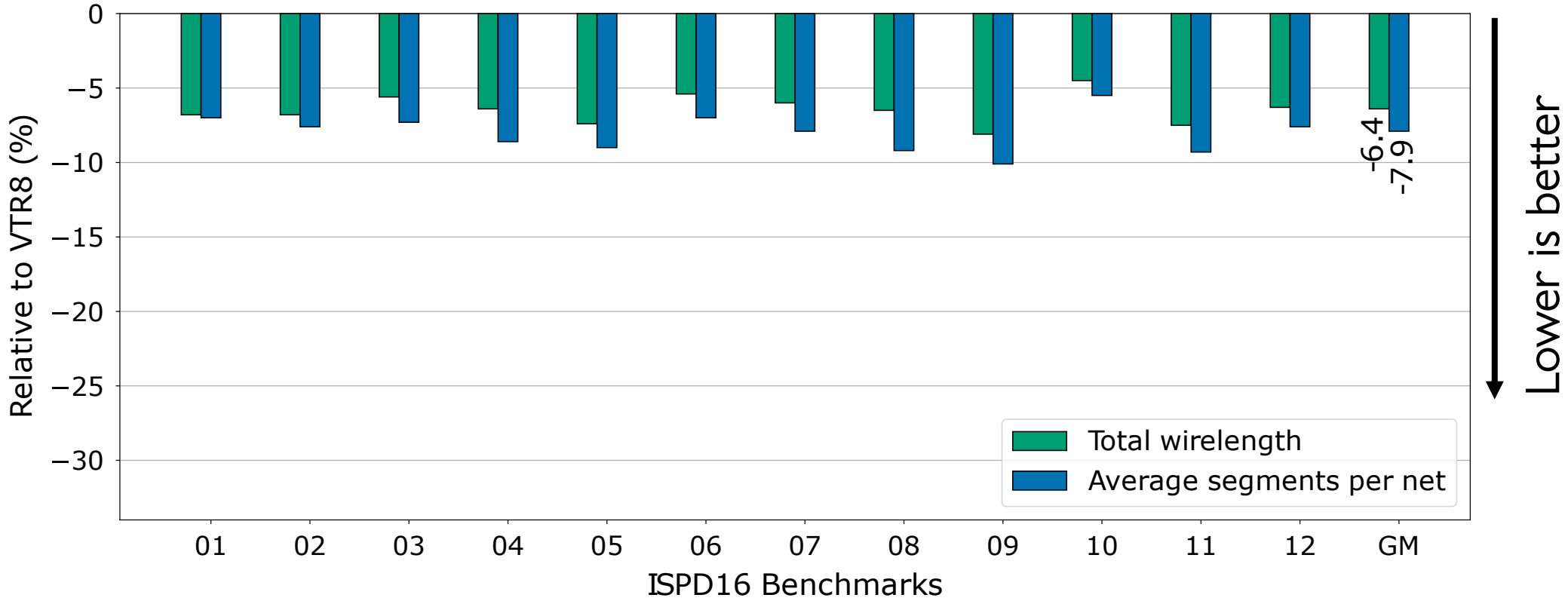
Experimental Setup

- Architecture: Closely resembling AMD UltraScale
- Benchmarks: ISPD16 (12 designs)
- Placements: UTPlaceF
- Router: Verilog-to-Routing 8
- Random sink shuffles: 48

Thread-level Parallelism

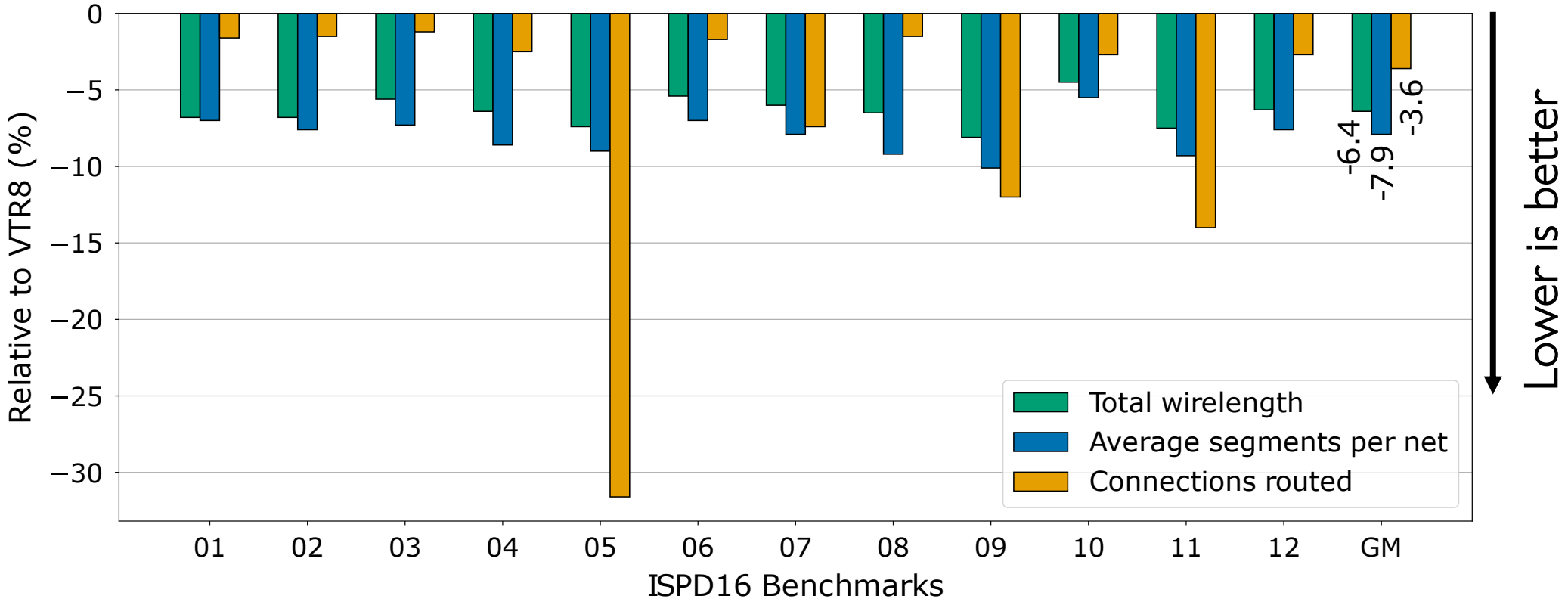


Results: Standard Setting



6.4% ↓ in total wirelength and 7.9% ↓ in average segments per net

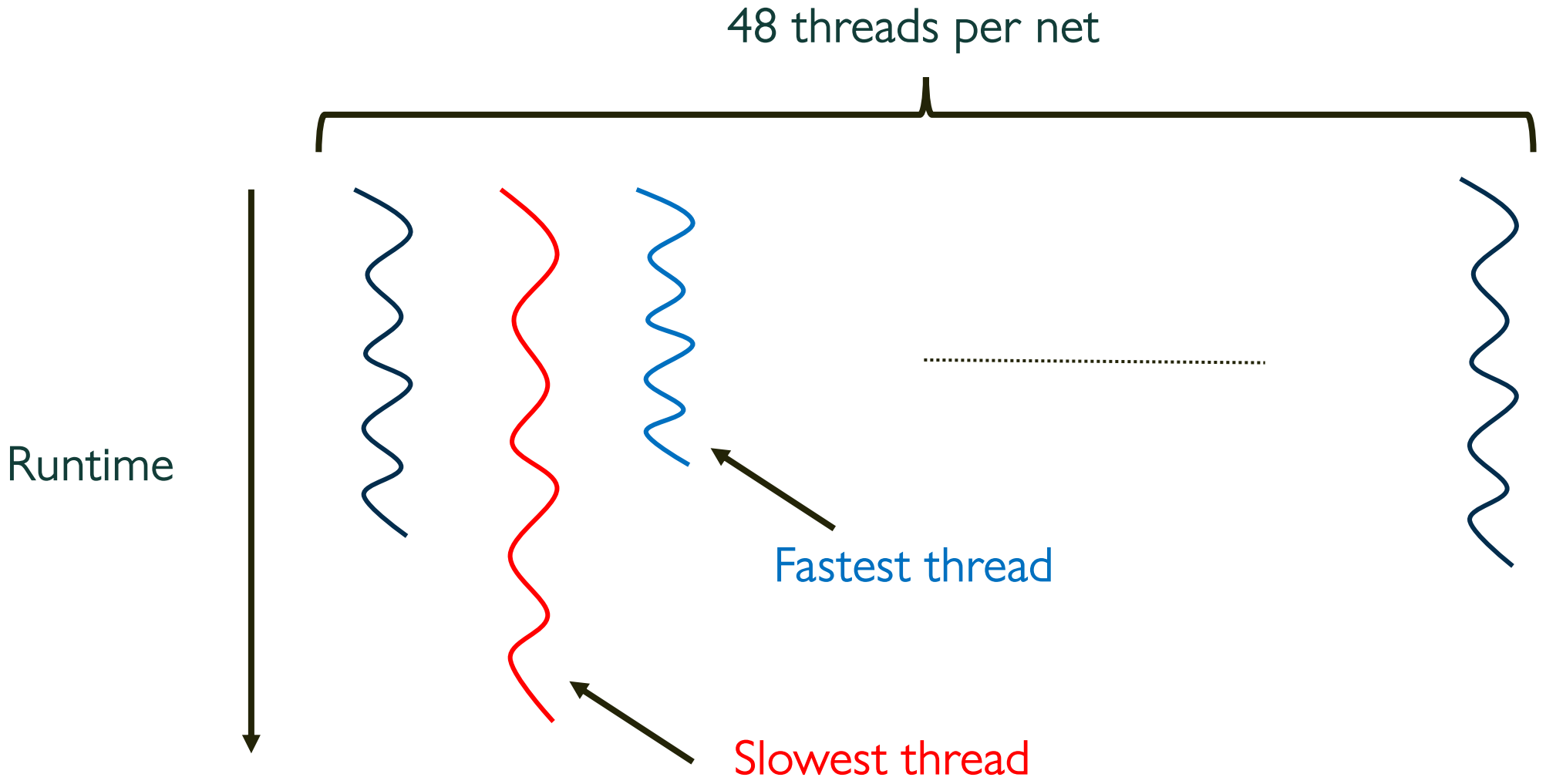
Results: Standard Setting



Reduction in total rerouted connections → congestion drops faster

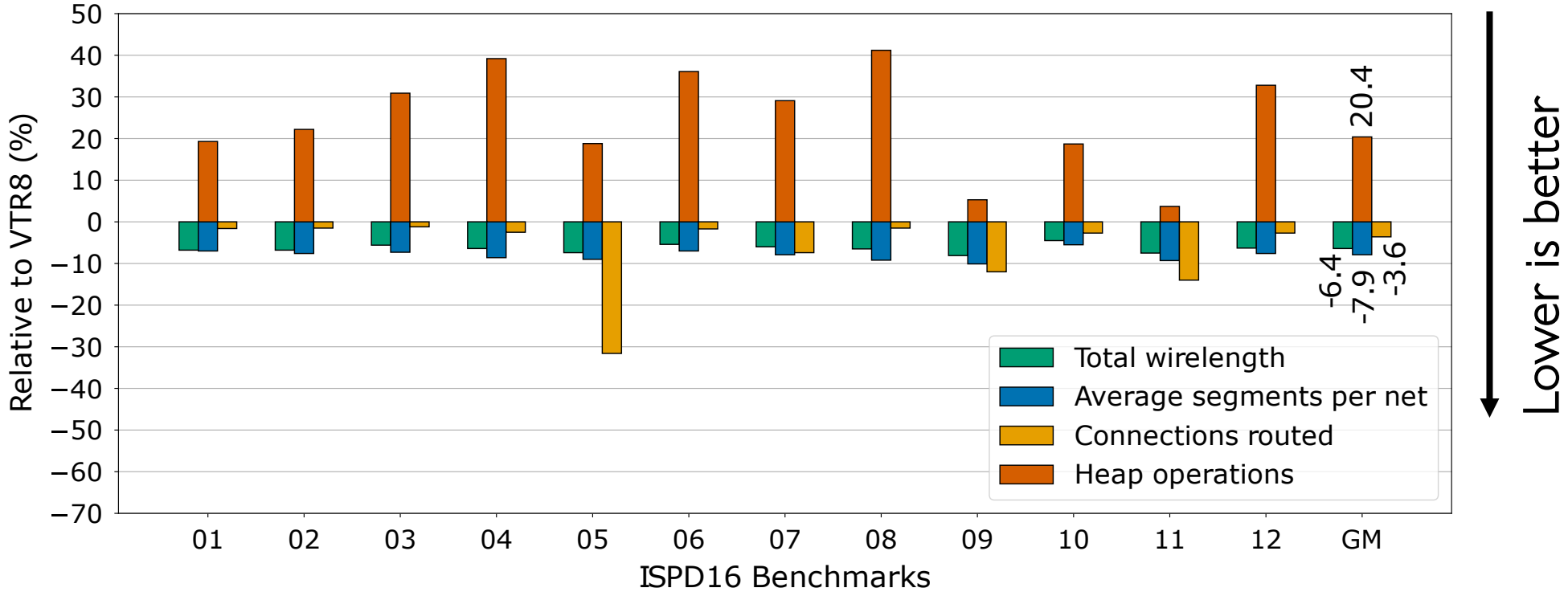
Impact on Runtime

Runtime dictated by slowest thread



Impact on Runtime

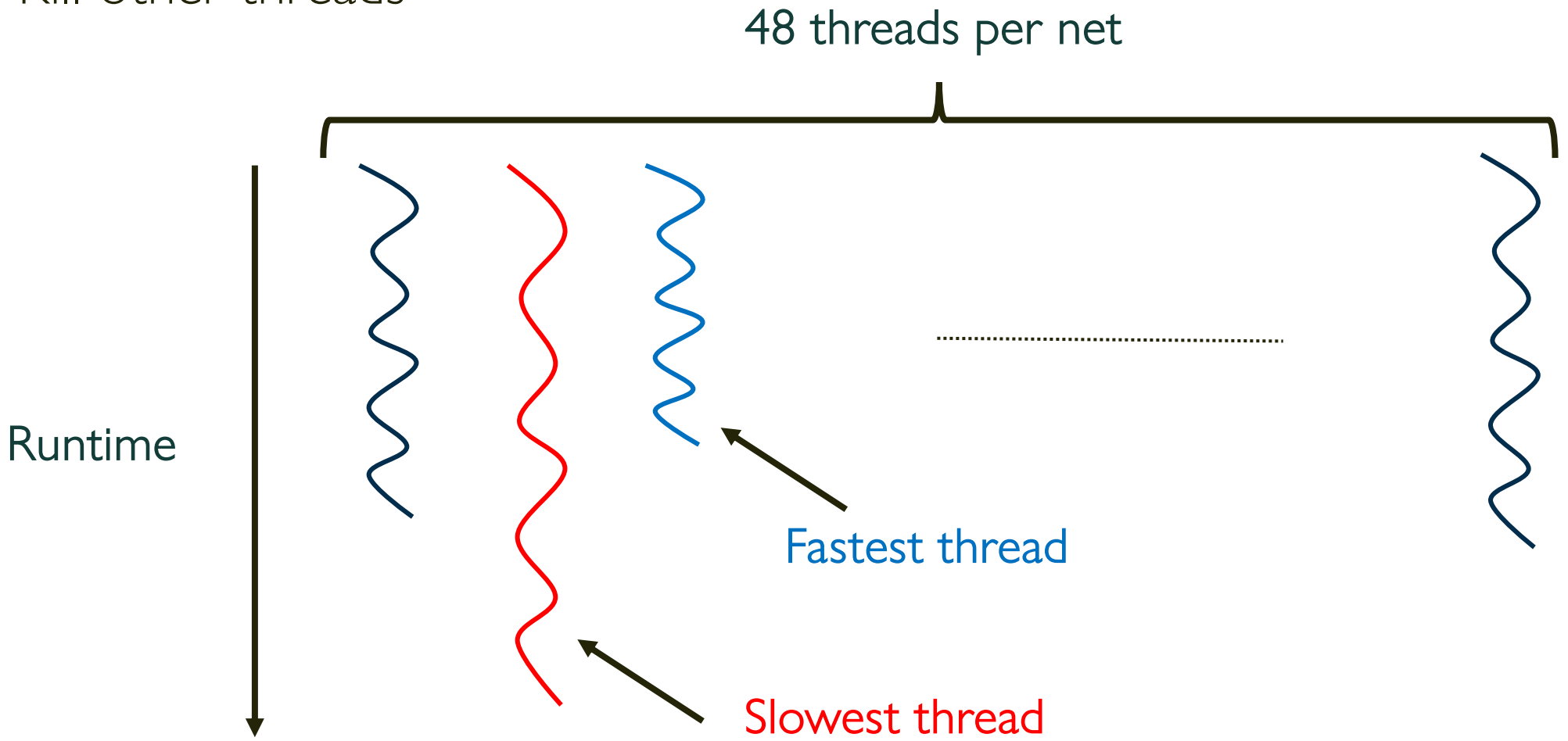
- Assuming no other runtime overheads
- As a proxy for runtime, we measure heap operations (machine agnostic)



By investing 20% ↑ heap operations, routed wirelength ↓ 6.4%

Runtime Co-optimization

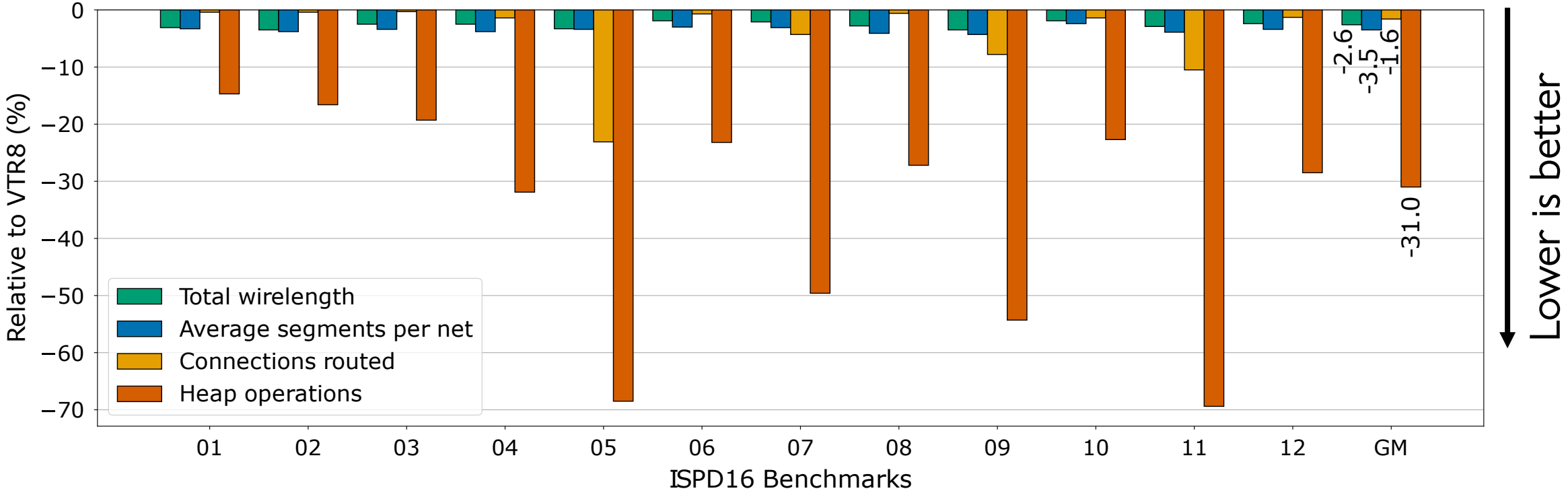
- Select the tree corresponding to fastest thread
- Kill other threads



Tree Selection Strategy for Runtime

Select a tree corresponding to fastest thread (lowest heap operations)

→ correlates with a tree having minimum number of nodes



Runtime improves by a tailored tree selection, preserving total wirelength

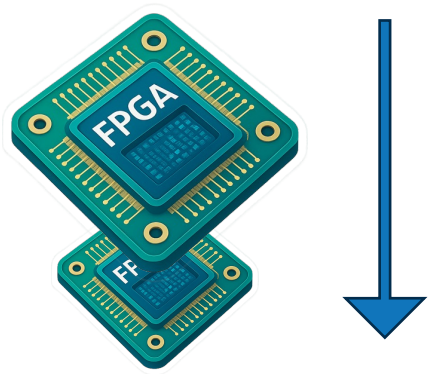
Conclusions

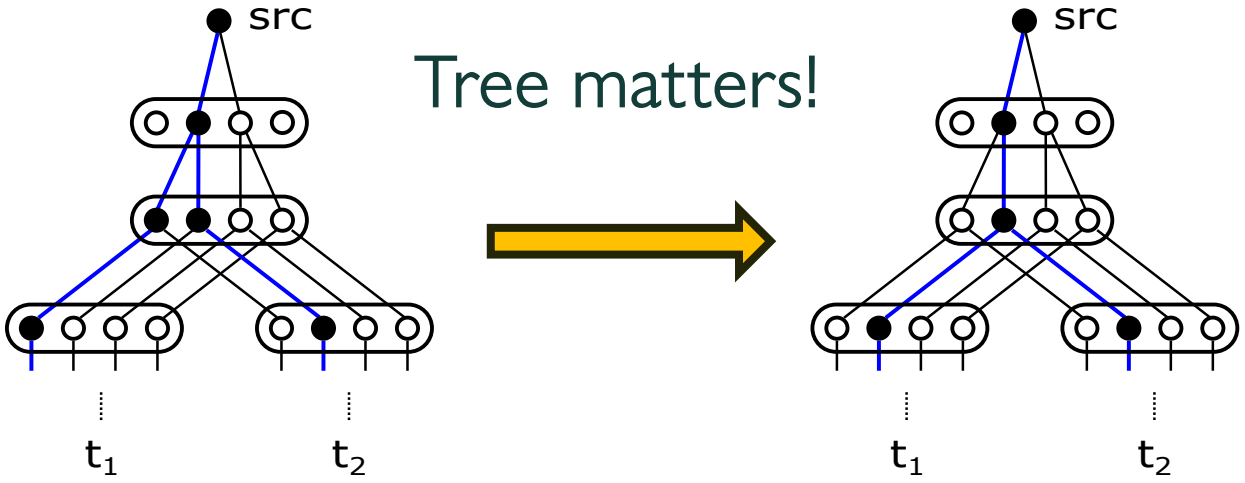
Utilize framework to diagnose fundamental inefficiencies

Artifacts

Develop algorithms for holistic route tree construction

Redesign FPGA routing architectures for silicon efficiency





Copyright and Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Conclusions

Utilize framework to diagnose fundamental inefficiencies

Artifacts

Develop algorithms for holistic route tree construction

Redesign FPGA routing architectures for silicon efficiency