An Open-source Lightweight Timing Model for RapidWright

Pongstorn Maidee, Chris Neely, Alireza Kaviani and Chris Lavin
Xilinx Research Labs
San Jose, CA, USA

FPT'19, December 12, 2019

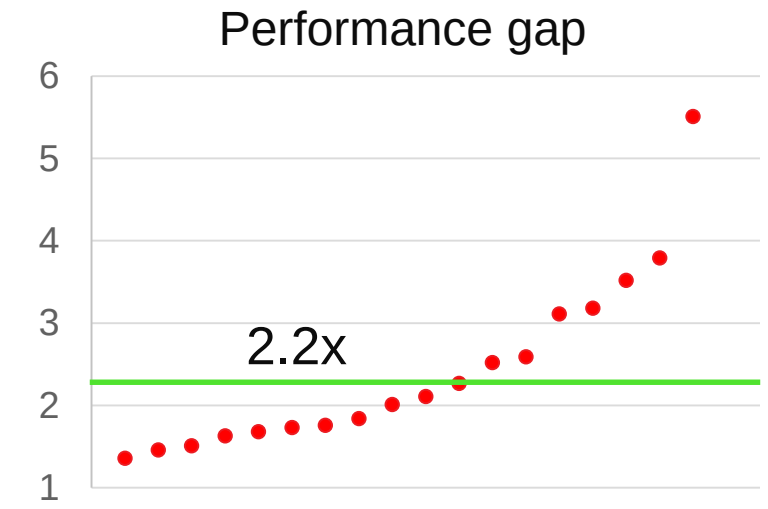
XILINX.

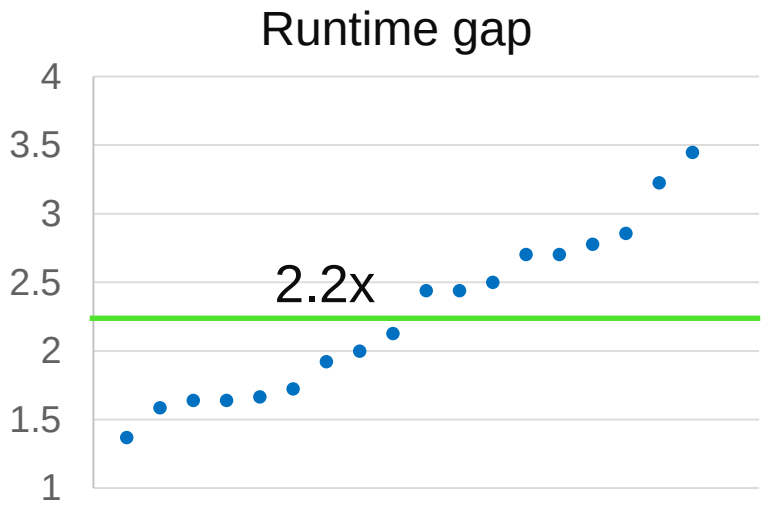
Outline

- Problem statement
- Background (RapidWright)
- Timing model derivation and usage
- Experimental results
- Conclusions and future work

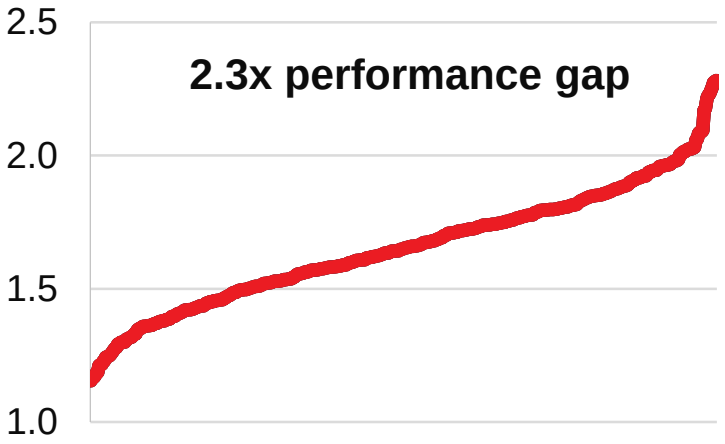
Open Source for FPGAs

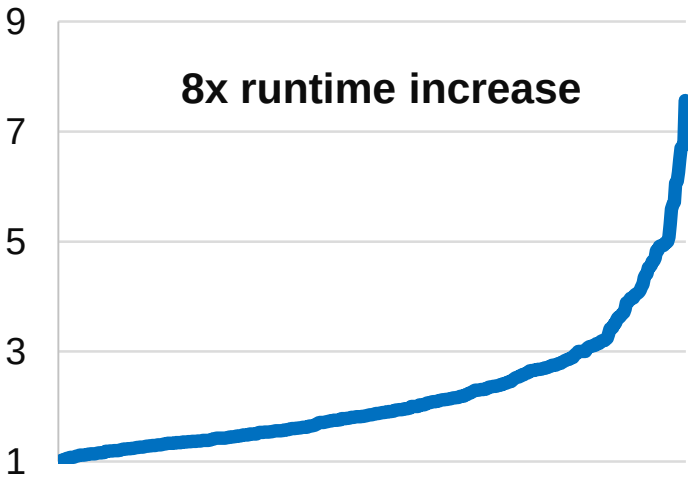
- Open source expedites software development
  - Expand academic ecosystem
  - Enable research collaboration
- Academic tools for FPGAs
  - VPR (FPL 1997), VTR (FPGA 2012)
  - Fostered many research activities
  - Large gap in figures of merit to commercial tools
    - 2.2x speed-performance, 2.2x runtime [FPT2015]
- Open source tools for commercial FPGAs
  - RapidWright, ICEStorm
  - Foundational functionalities (without timing model)





Importance of Timing-driven Tools





- Without timing model
  - Depth-based optimization
    - Inaccurate
  - Delegate to Vivado
    - Increase runtime

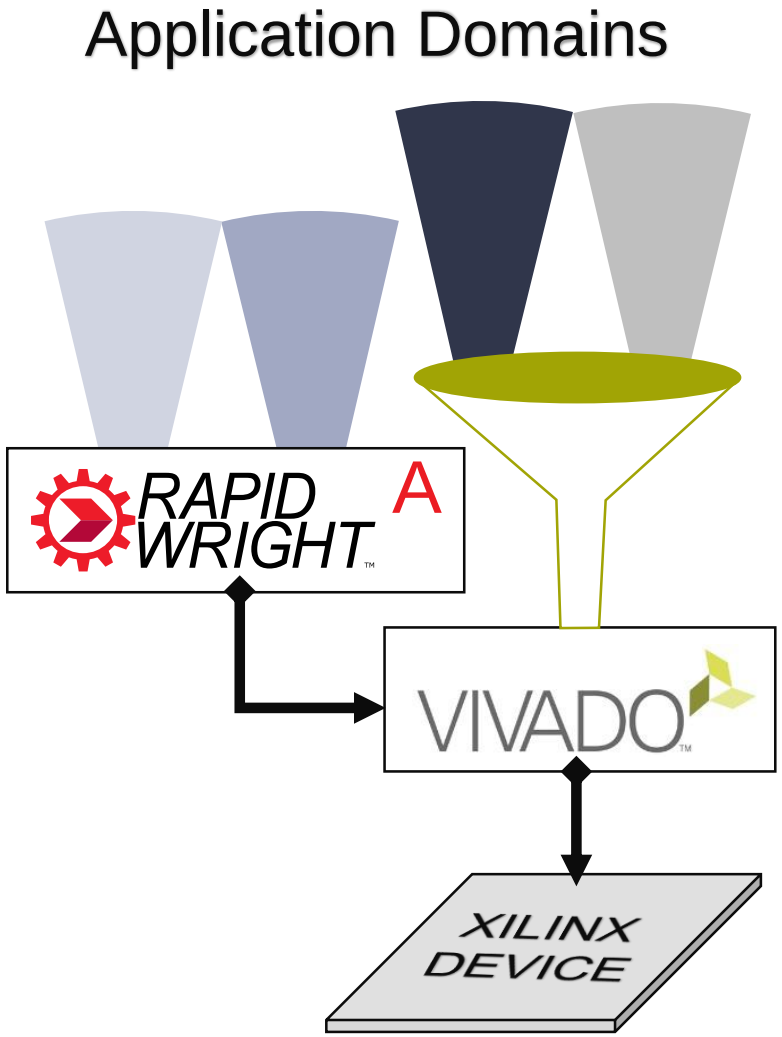
- An open timing model
  - More flexibility for customized backend solutions
  - Minimizes Vivado timing iterations
  - Explores wider solution space

Why RapidWright?

Vivado must ***generalize***

Why RapidWright?

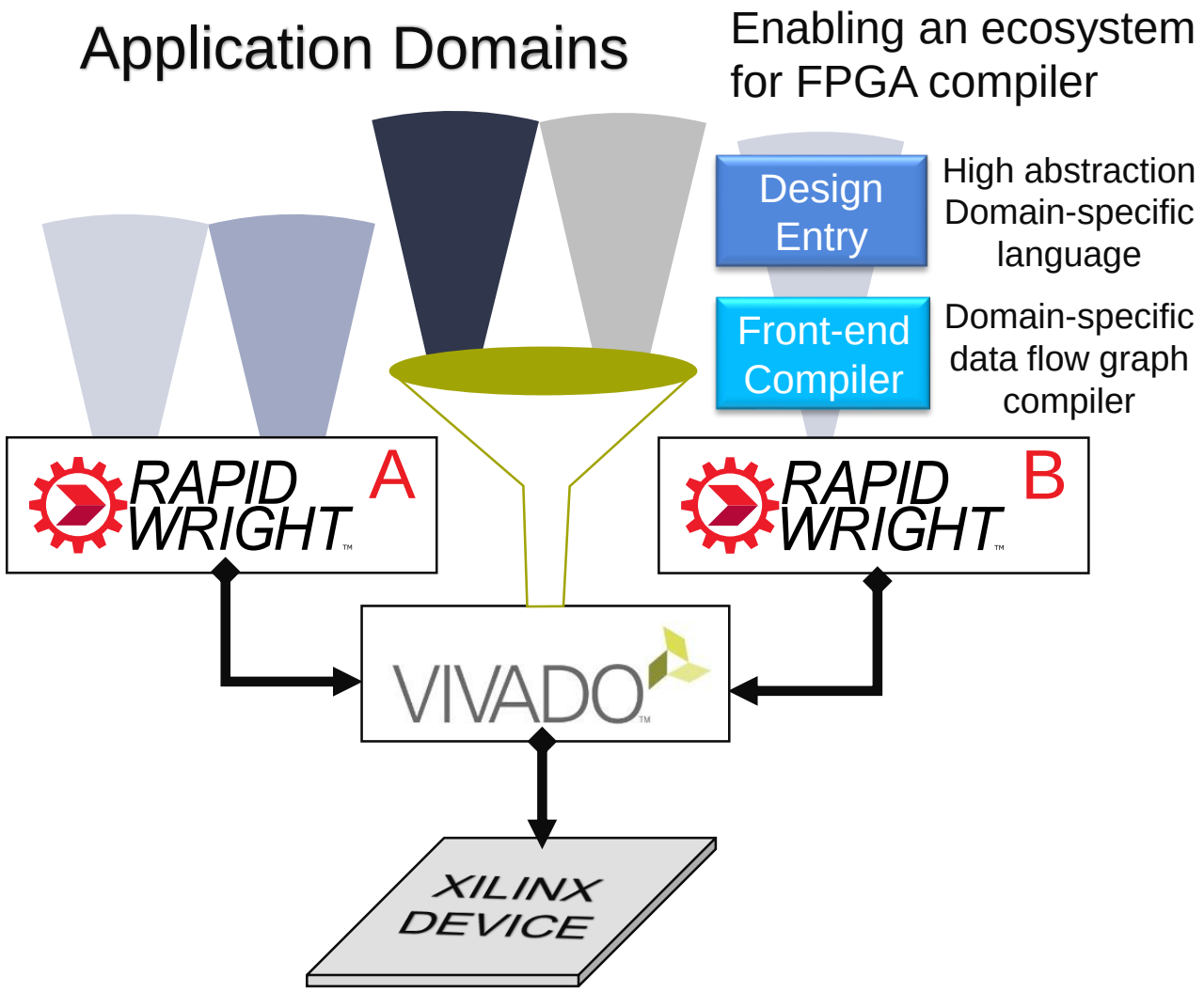
Vivado must ***generalize***



RapidWright can ***specialize***

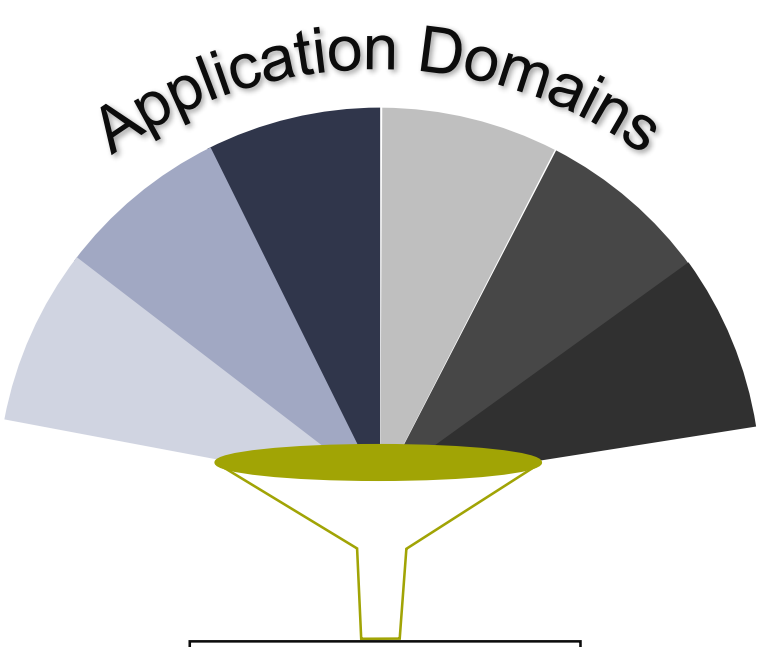
Why RapidWright?

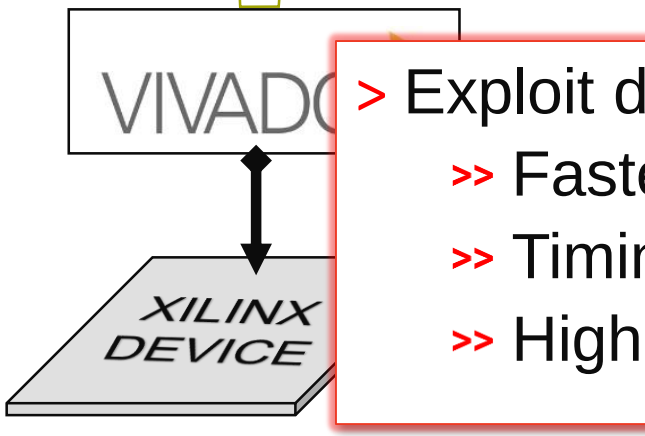
Vivado must ***generalize***



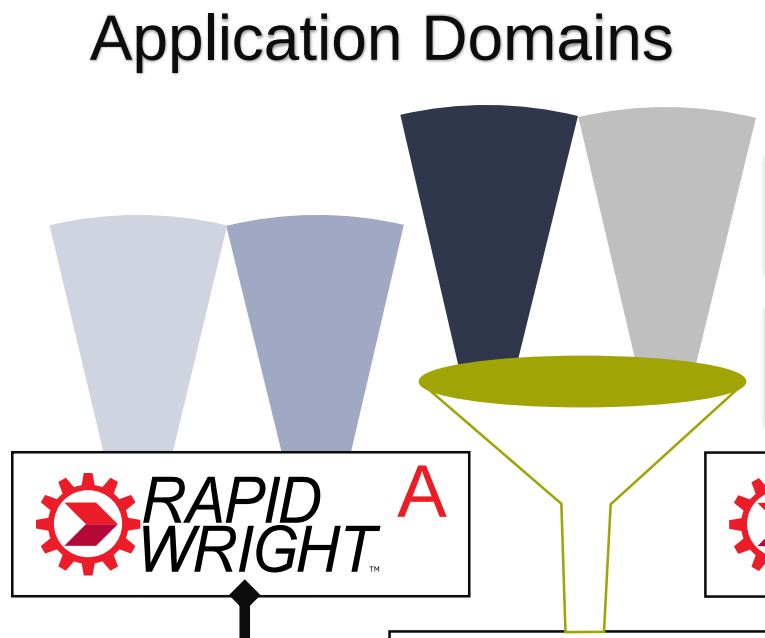
RapidWright can ***specialize***

Why RapidWright?





Vivado must ***generalize***



Enabling an ecosystem for FPGA compiler

Design Entry — High abstraction Domain-specific language

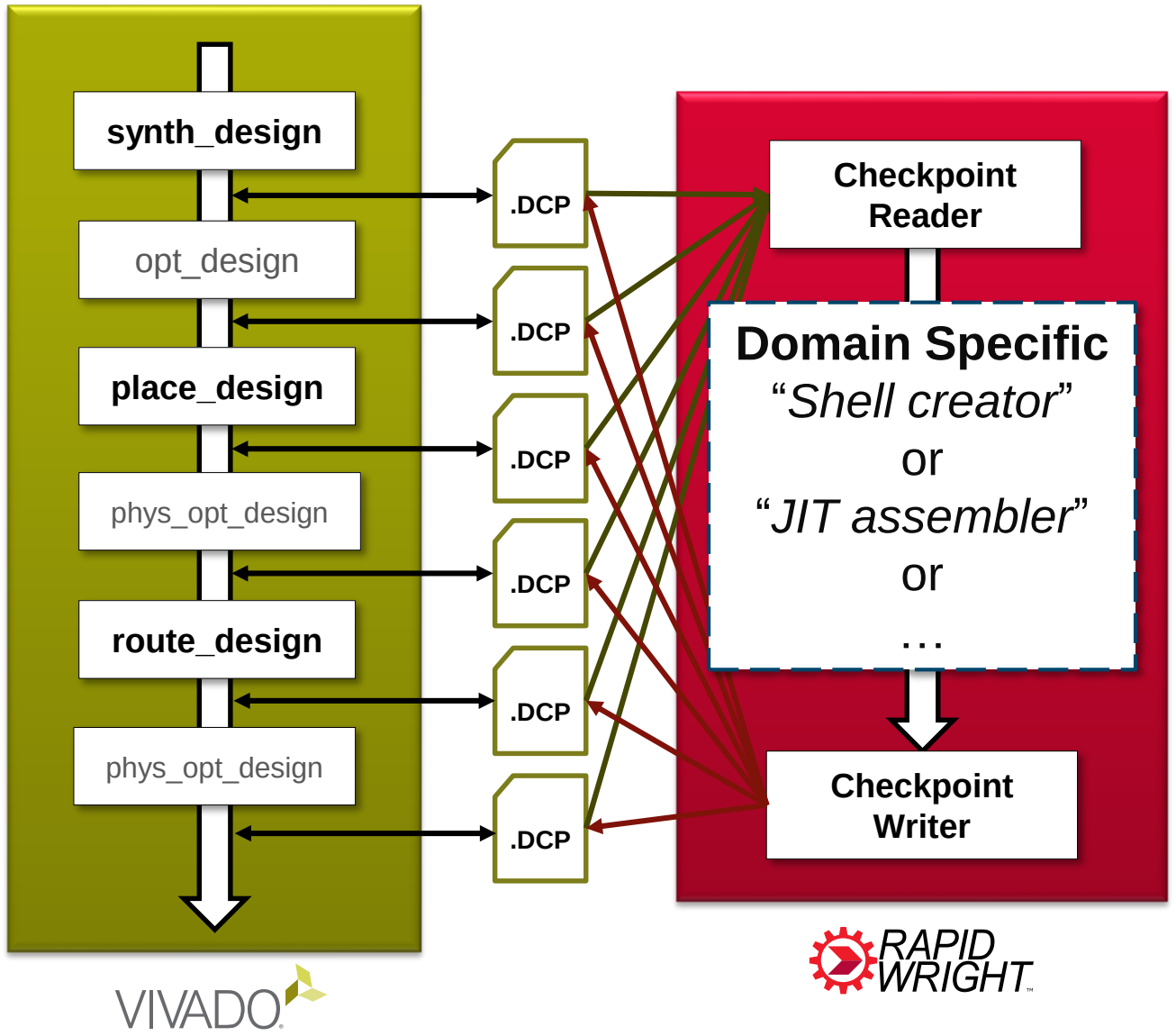
Front-end Compiler — Domain-specific data flow graph compiler

RapidWright can ***specialize***

> Exploit domain-specific traits for:
>> Faster compile time
>> Timing closure predictability
>> Higher performance

RapidWright Overview

- **Companion framework for Vivado**
  - Fast, light-weight, open source
  - Java code, Python scripting
  - Communicates through Design CheckPoints[1] (DCPs)
- **Enables targeted solutions**
  - Reuse & relocate timed pre-implemented modules
- **Vivado dependency for timing poses an obstacle for timing driven algorithms**



[1]DCP = netlist + P&R data + constraints

An Open Timing Model for RapidWright

Goals:

1) Fast and lightweight footprint
2) Sufficient accuracy
3) General and extensible

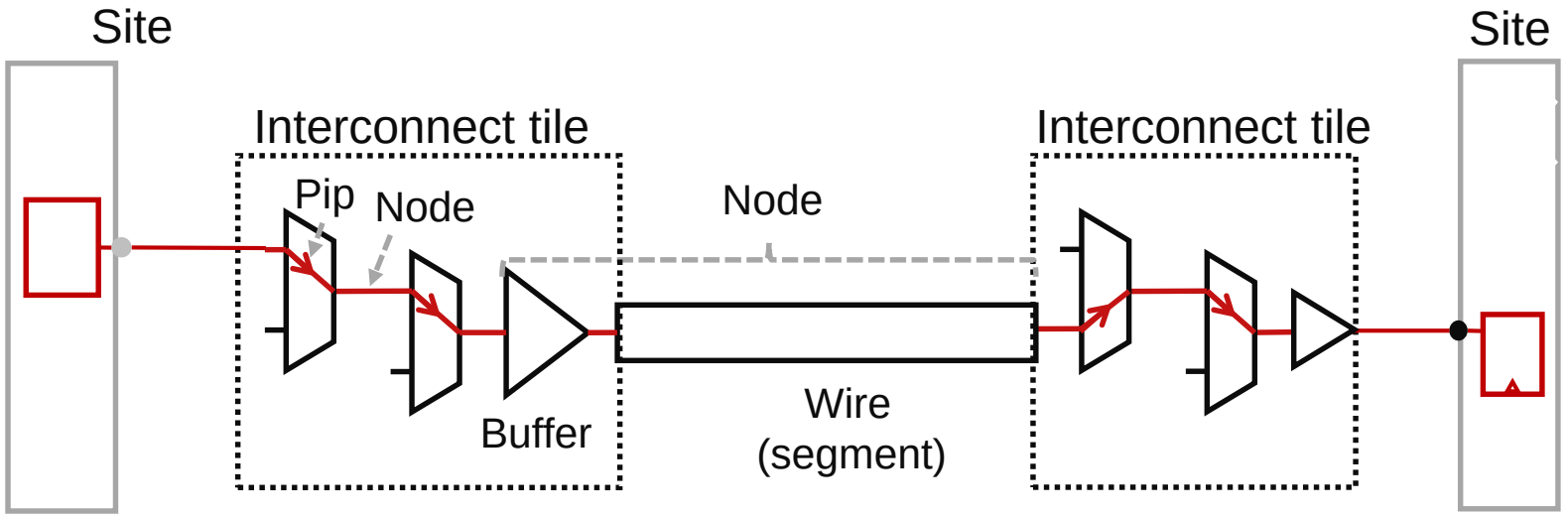
An Open Timing Model for RapidWright

Goals:

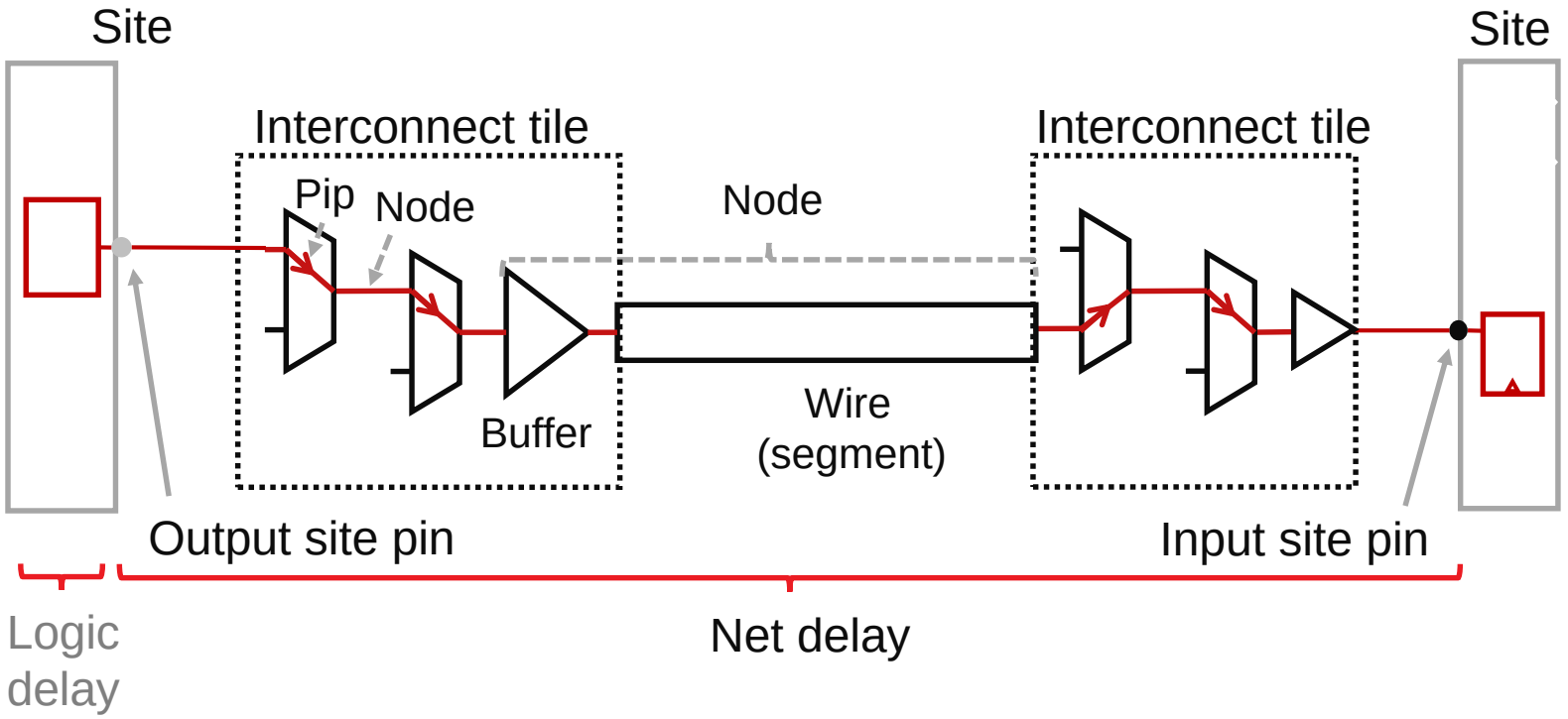
1) Fast and lightweight footprint
2) Sufficient accuracy
3) General and extensible

	Vivado	RapidWright
Abstraction	>> ~4k switches in one tile >> ~30k tiles in a small UltraScale+ device	>> ~100 timing groups (TGs)
Memory footprint	>> Store delay per resource	>> Equation-based

Abstracting Routing Fabric Delays

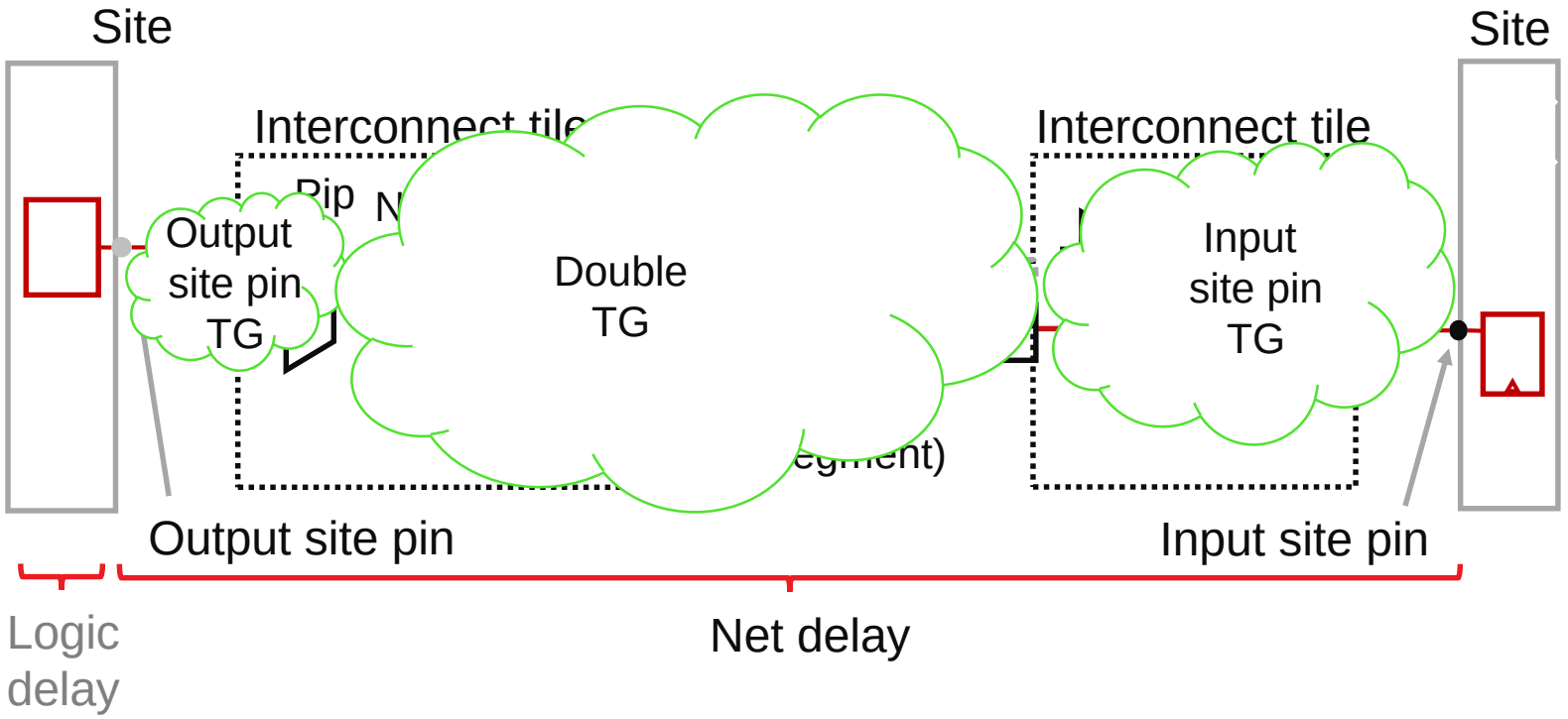


Abstracting Routing Fabric Delays



> Net delay = Σ pip delay + Σ node delay

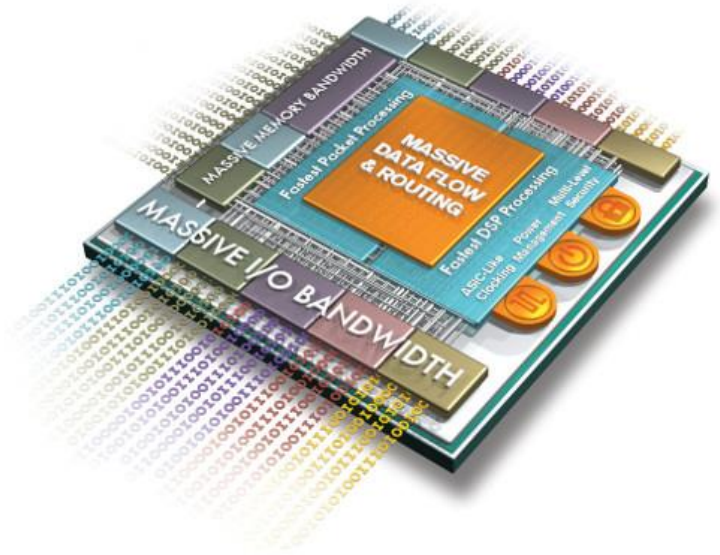
Abstracting Routing Fabric Delays



> Net delay = Σ pip delay + Σ node delay

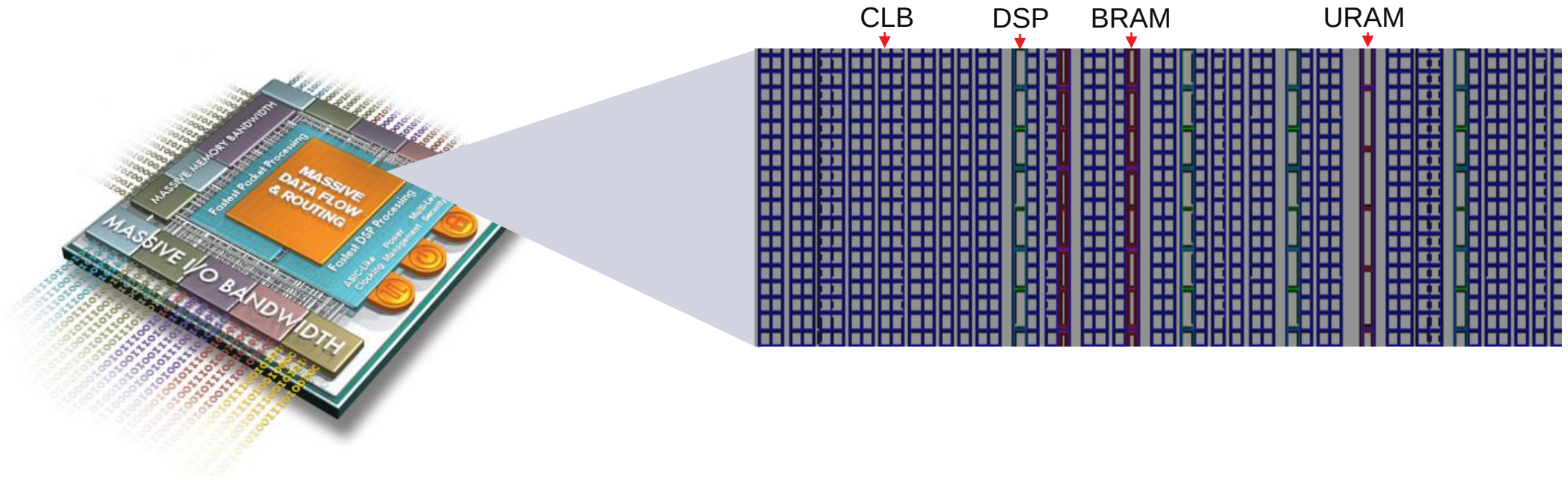
> Net delay = Σ TG's delay

Fabric Discontinuities



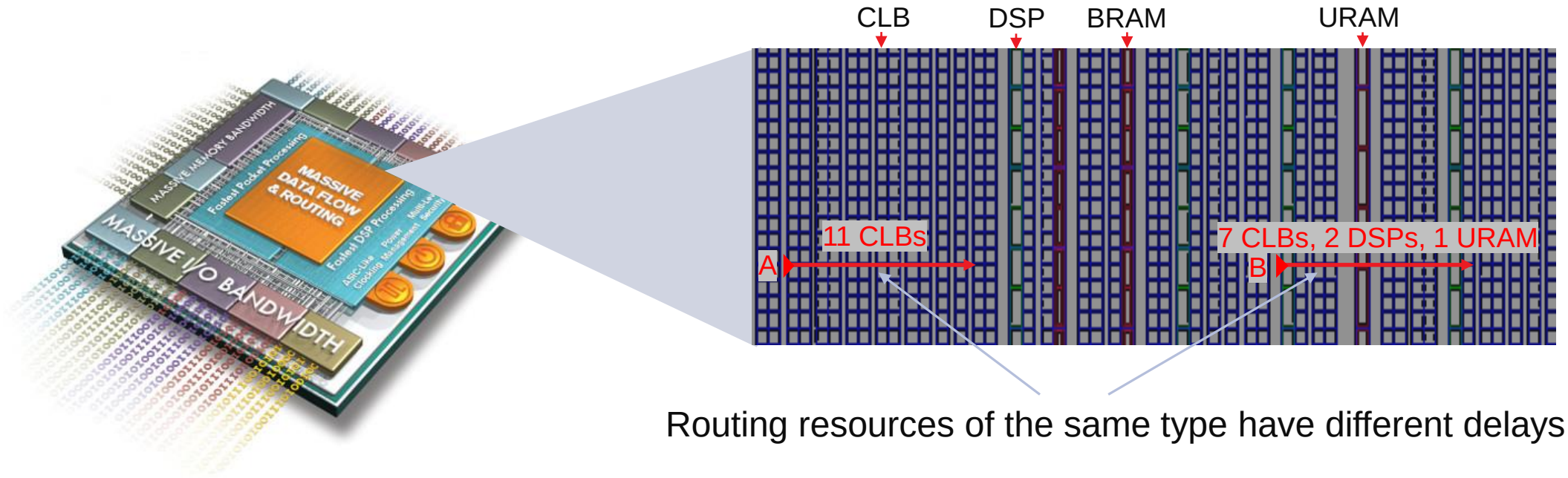
- **FPGA is heterogeneous: Transceiver, PCI, Processor Subsystem, CMAC**

Fabric Discontinuities



- FPGA is heterogeneous: Transceiver, PCI, Processor Subsystem, CMAC
- FPGA architecture is columnar and heterogeneous too! DSP, BRAM, URAM, IO

Fabric Discontinuities



Routing resources of the same type have different delays

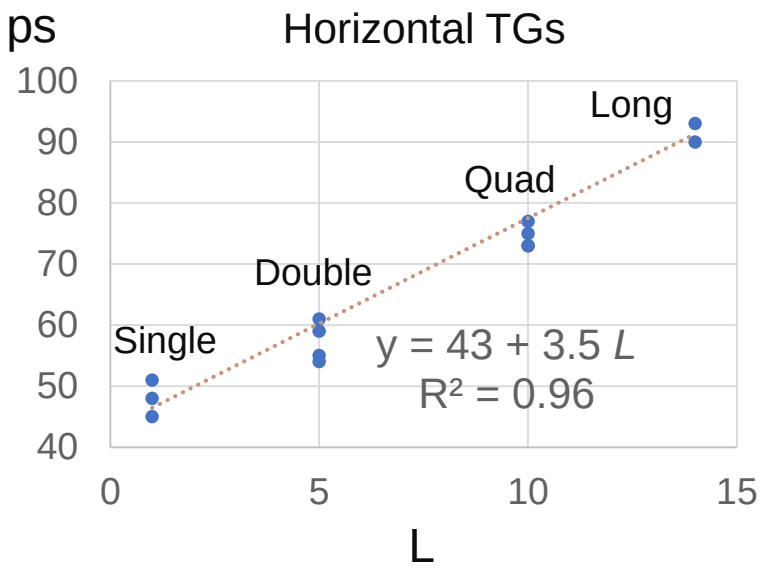
- **FPGA is heterogeneous: Transceiver, PCI, Processor Subsystem, CMAC**
- **FPGA architecture is columnar and heterogeneous too! DSP, BRAM, URAM, IO**
- **Heterogeneous nature causes timing discontinuity**

Equation-based Representation of Timing Group Delay

- TG's delay $\boldsymbol{delay = k_0 + k_1 L + k_2 d}$
  - L : nominal length of the segment
  - d : distance of crossing large tiles, such as DSP, BRAM, URAM, PCI and IO

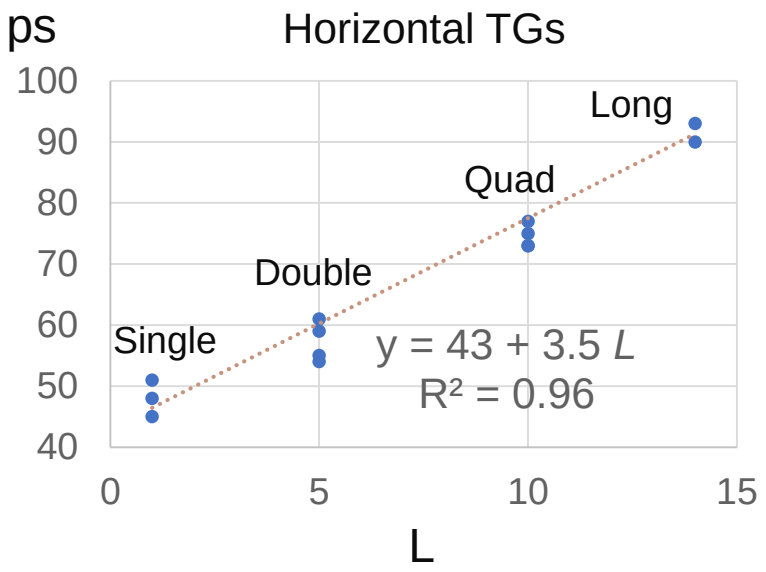
Equation-based Representation of Timing Group Delay

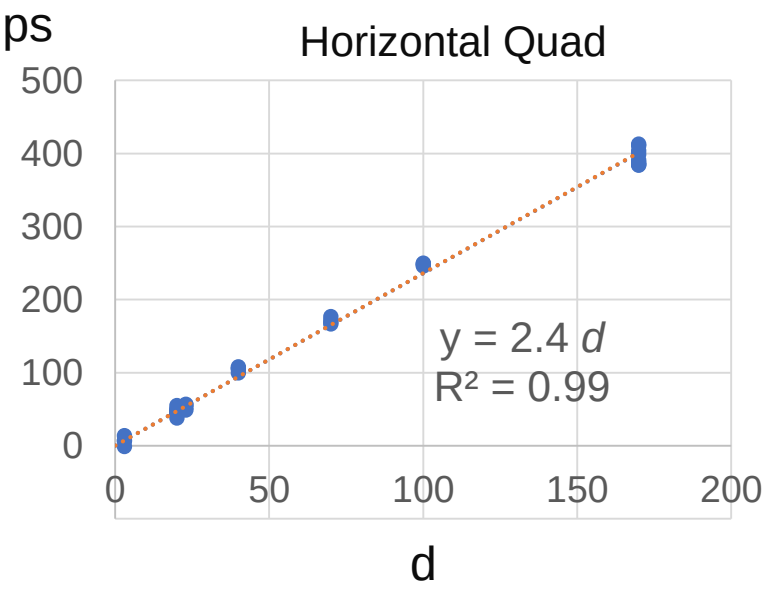
- TG's delay $\boldsymbol{delay = k_0 + k_1 L + k_2 d}$
  - L : nominal length of the segment
  - d : distance of crossing large tiles, such as DSP, BRAM, URAM, PCI and IO
- Find k_0, k_1, L
  - Plot delays of TG having $d = 0$
  - Adjust L so that all delays fit to a linear equation



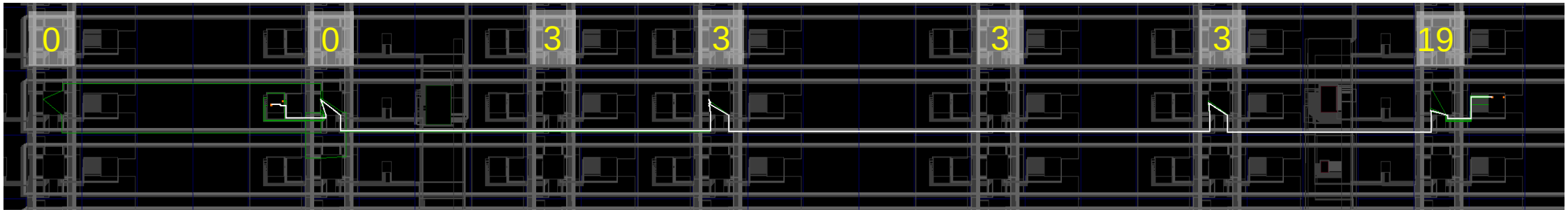
Equation-based Representation of Timing Group Delay

- TG's delay $\boldsymbol{delay = k_0 + k_1 L + k_2 d}$
  - L : nominal length of the segment
  - d : distance of crossing large tiles, such as DSP, BRAM, URAM, PCI and IO
- Find k_0, k_1, L
  - Plot delays of TG having $d = 0$
  - Adjust L so that all delays fit to a linear equation
- Find d for each TG type
  - Plot extra delays of TGs having $d > 0$
  - Adjust d of hard block columns to fit a linear equation



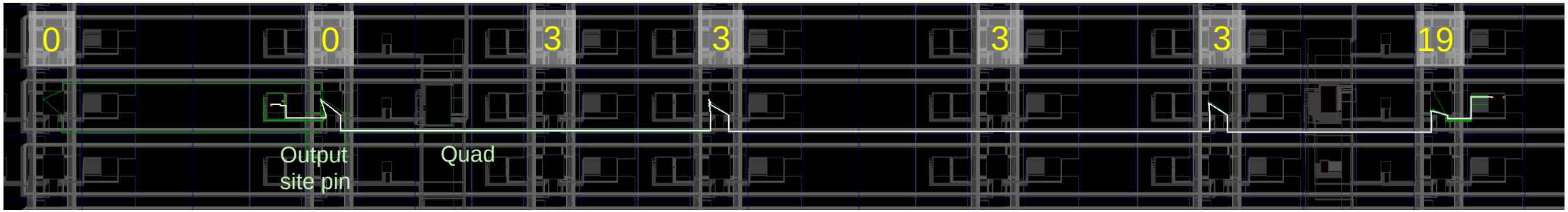


Example Inter-site Net Delay Computation



- Compute and store cumulative *d* on INT tiles (Once)
- Linearly traverse the route
- Recognize TG to select equation and *L*
- Lookup *d* and compute TG delay

Example Inter-site Net Delay Computation

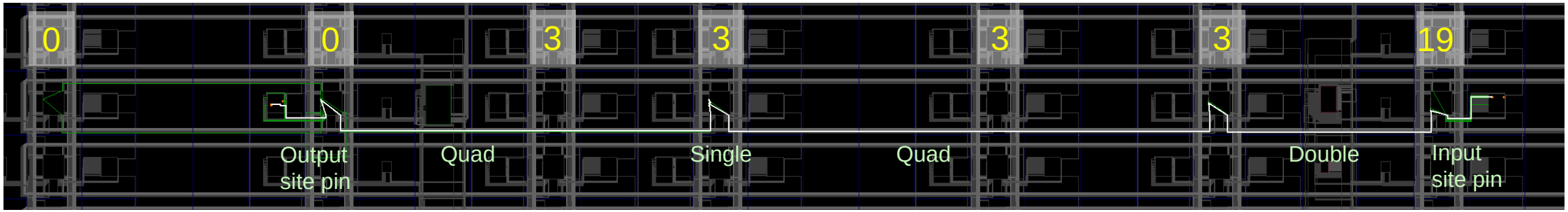


0 + 85.2

$$43 + 3.5 \times 10 + 2.4 \times 3$$
$$k_0 \quad k_1 \quad L \quad k_2 \quad d$$

- Compute and store cumulative *d* on INT tiles (Once)
- Linearly traverse the route
- Recognize TG to select equation and *L*
- Lookup *d* and compute TG delay

Example Inter-site Net Delay Computation



$$346.5 = 0 + 85.2 + 43 + 78 + 97.3 + 43$$

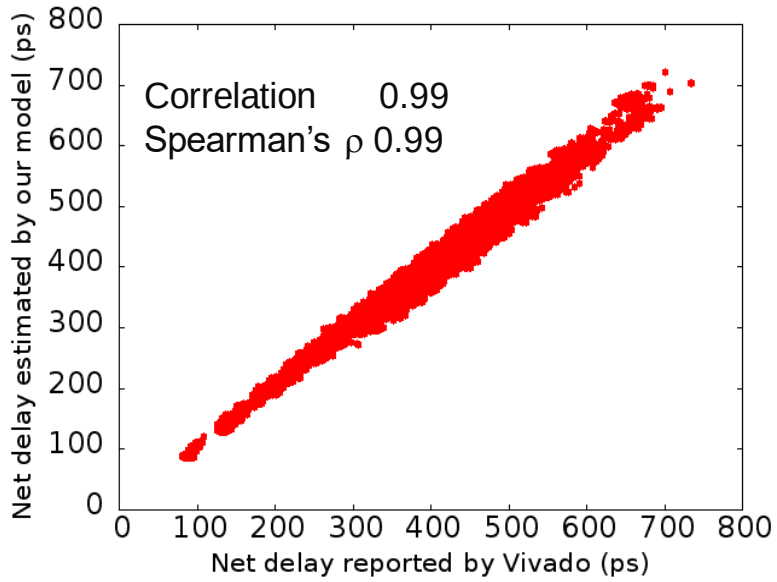
$$43 + 3.5 \times 10 + 2.4 \times 3 \qquad 43 + 3.5 \times 10 + 2.4 \times \mathbf{0} \qquad 43 + 3.5 \times 5 + 2.3 \times 16$$

$$k_0 \quad k_1 \quad L \quad k_2 \quad d$$

- Compute and store cumulative *d* on INT tiles (Once)
- Linearly traverse the route
- Recognize TG to select equation and *L*
- Lookup *d* and compute TG delay

Timing Model Correlation

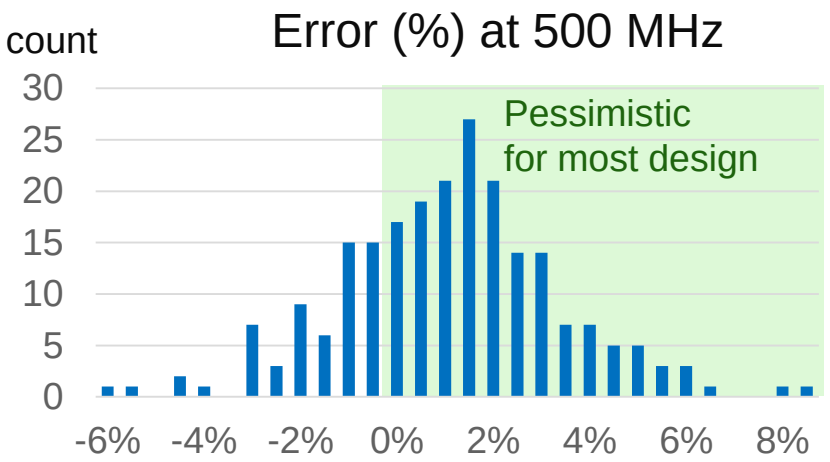
- Randomly placed 2-pin net in one clock region
- Repeat 40K times
- High correlation
- Pessimistic on average at 0.7%

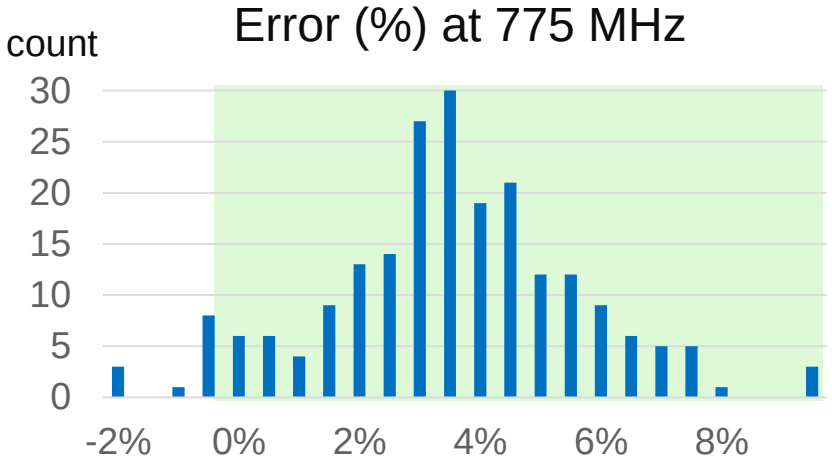


Estimation error

	Error (ps)	Error %
Avg	0.8	0.7
Min	-53.9	-12.7
Max	40.0	14.9

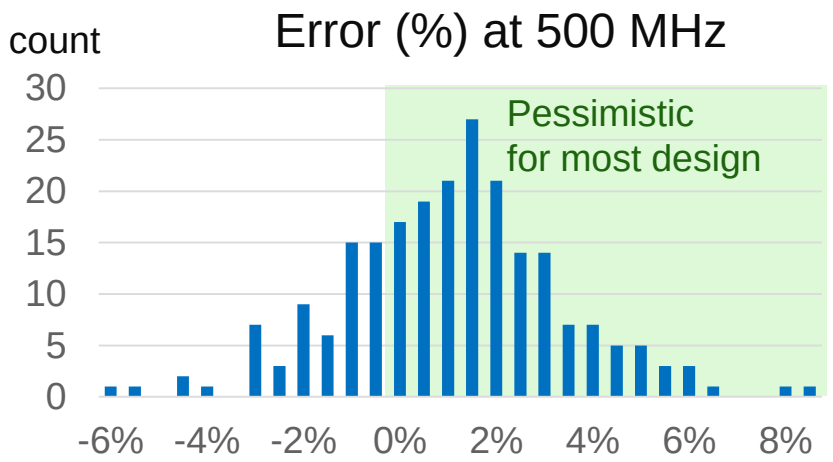
Timing Model Accuracy

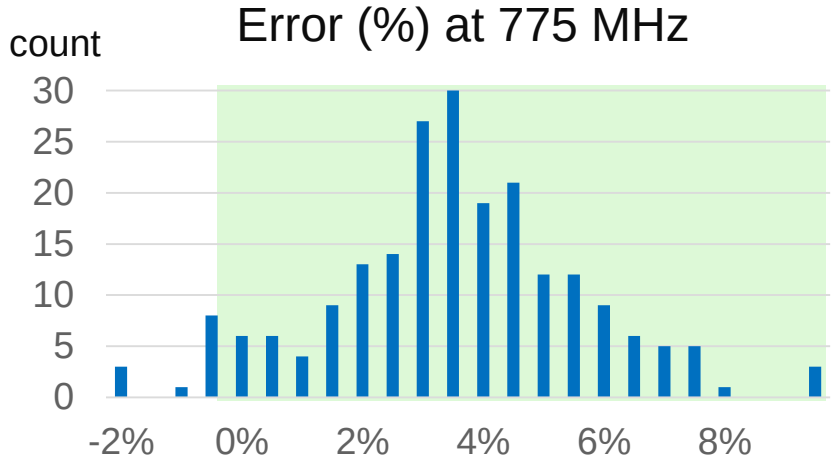


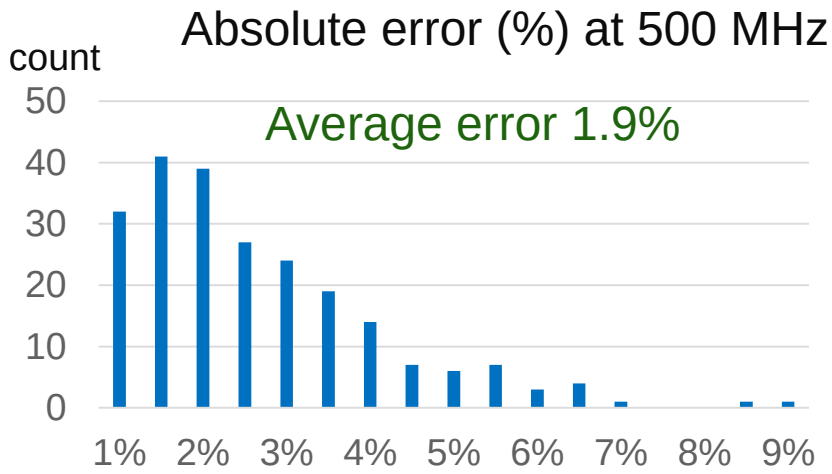


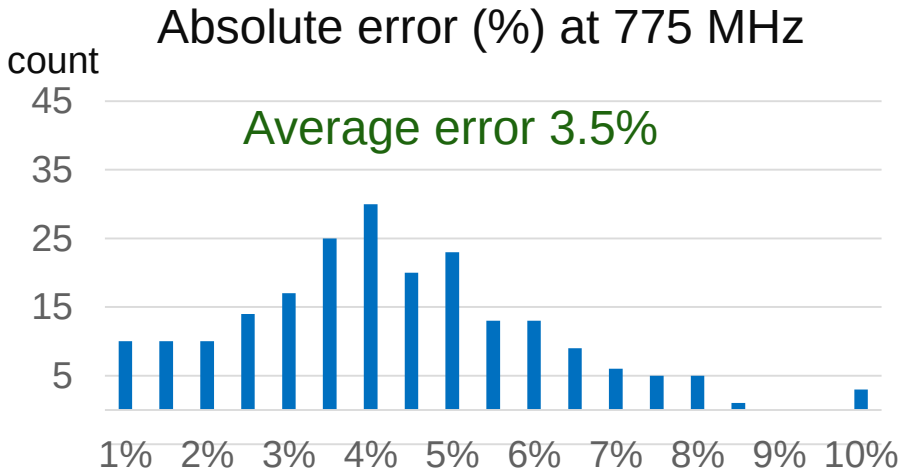
- 240 Synthetic designs
  - LUT utilization: 50% - 88%, Rent's exponent: 0.3 – 0.8
  - Logic depth: 3 to 7

Timing Model Accuracy









- 240 Synthetic designs
  - LUT utilization: 50% - 88%, Rent's exponent: 0.3 – 0.8
  - Logic depth: 3 to 7

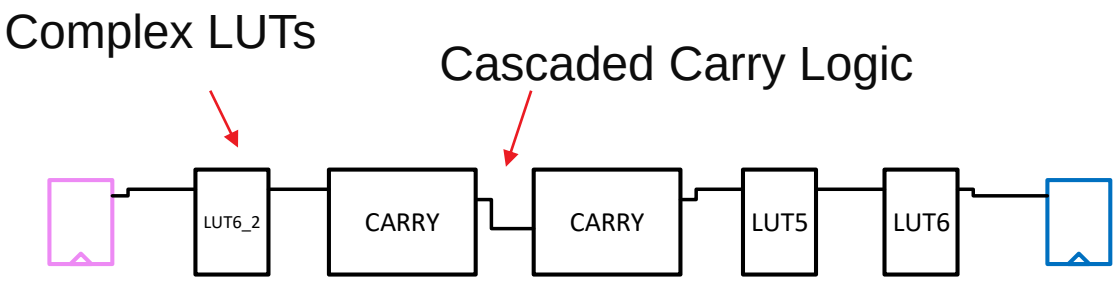
Real Designs (Data Path Delay)

Design	Vivado (ps)	RapidWright (ps)	Estimating error
Microblaze	1874	1906	1.7%
Picoblaze	1609	1615	0.4%
Parser	2742	2665	-2.8%

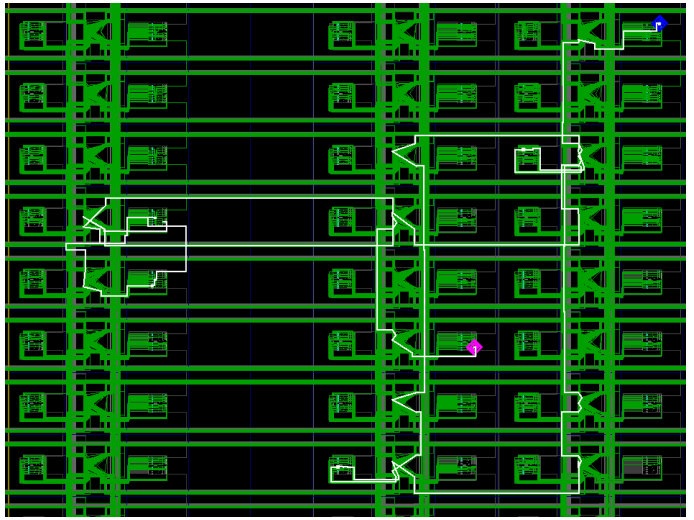
Real Designs (Data Path Delay)

Design	Vivado (ps)	RapidWright (ps)	Estimating error
Microblaze	1874	1906	1.7%
Picoblaze	1609	1615	0.4%
Parser	2742	2665	-2.8%

Non-trivial critical path

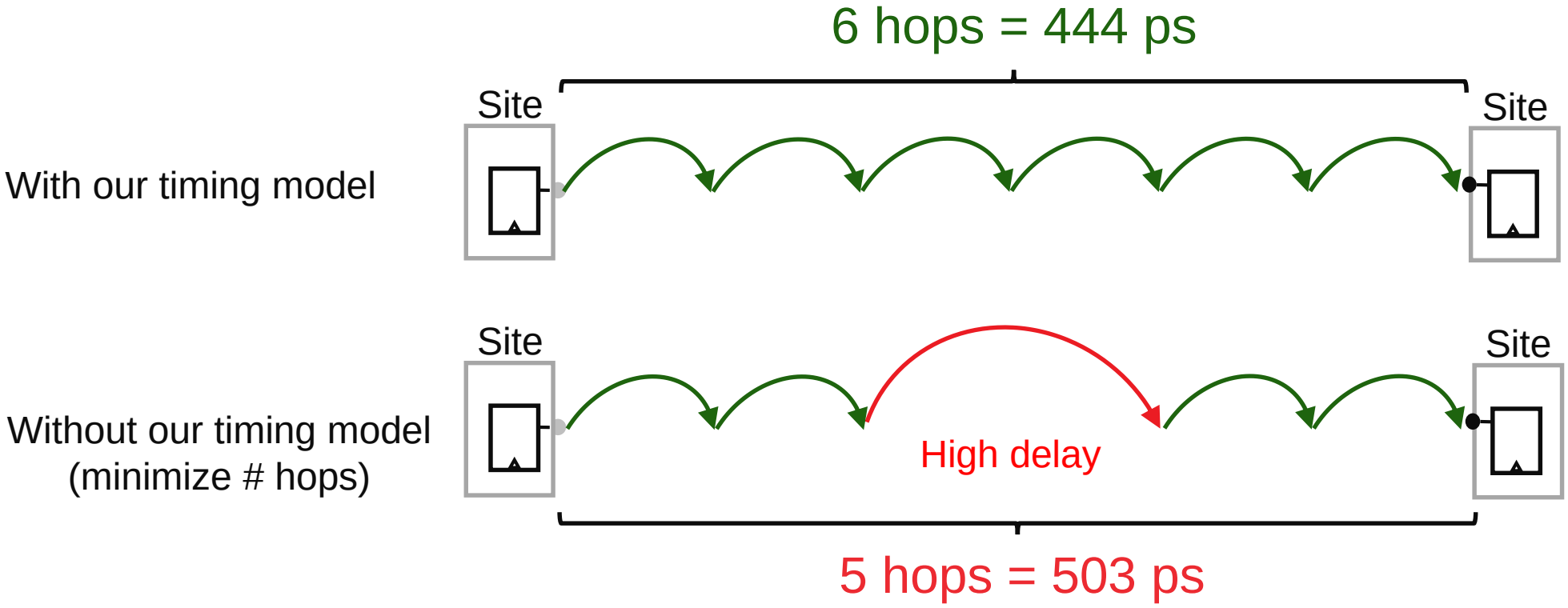


Complex routing



Timing-driven Routing

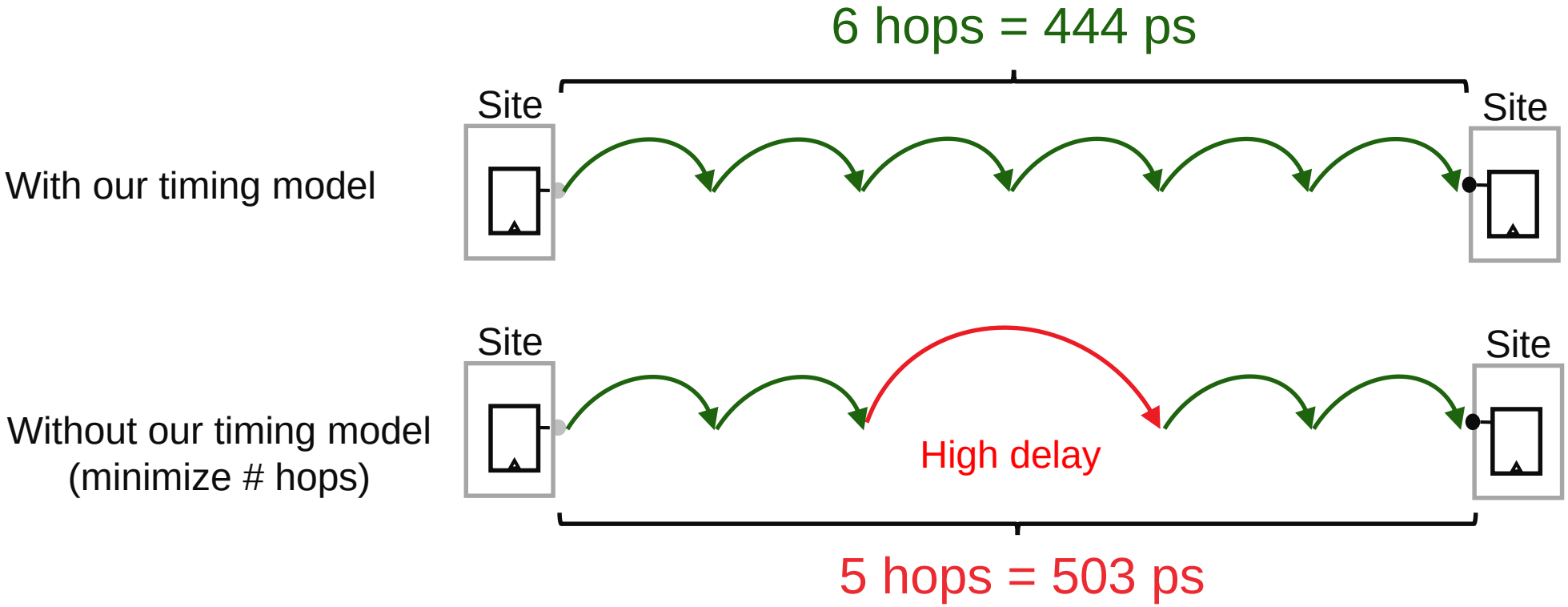
> Pin-to-pin router (for demonstration)



Route	Timing (ps)	Non-timing (ps)	Improved
Horizontal →	444	503	11.7%

Timing-driven Routing

- Pin-to-pin router (for demonstration)



Route	Timing (ps)	Non-timing (ps)	Improved	Vivado (ps)
Horizontal →	444	503	11.7%	429
Diagonal ↗	704	813	13.4%	706

Conclusions and Future Work

- Introduce a lightweight timing model
  - 2% accuracy on average compared to Vivado's result
  - Small memory footprint (less than 1 KB)

- Enable timing-driven backend tools in RapidWright
  - Available **NOW** at http://www.rapidwright.io
  - Timing library Java package: com.xilinx.rapidwright.timing
  - Examples
    - http://www.rapidwright.io/docs/ReportTimingExample.html
    - http://www.rapidwright.io/docs/PipelineGeneratorExampleWithRouting.html

- Future work
  - Expand coverage to DSP and BRAM
  - Model clock network

Looking forward to contributions from the community