Post-Placement Timing Optimisations on Synchronous & Asynchronous Designs under Power Constraints

Dimitrios Tsalapatas

University of Thessaly, Department of Electrical and Computer Engineering

Funded by the European Union

UK Research and Innovation

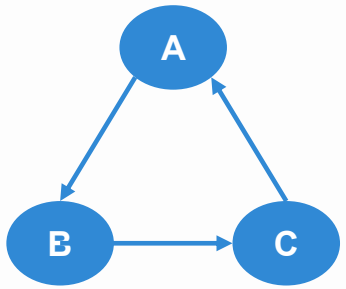
Funded by the European Union (Grant Agreement № 101160314). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Univ. of Manchester is funded through the Horizon Europe Guarantee Funding Scheme from UKRI (United Kingdom Research and Innovation)

Motivation: Two Timing Problems

- **Asynchronous designs (the novel contribution)**
  - Combinational feedback loops break Static Timing Analysis — no valid startpoint to begin delay propagation
  - No established post-placement optimization flow exists for these cyclic timing paths
- **Synchronous designs (generalizing the technique)**
  - Standard STA works fine here — but commercial resizers still leave delay and power on the table under tight constraints
  - The same lightweight upsize / buffer engine we built for cycles applies directly to ordinary critical paths

Cyclic graph — async



✗ **Standard STA cannot start**

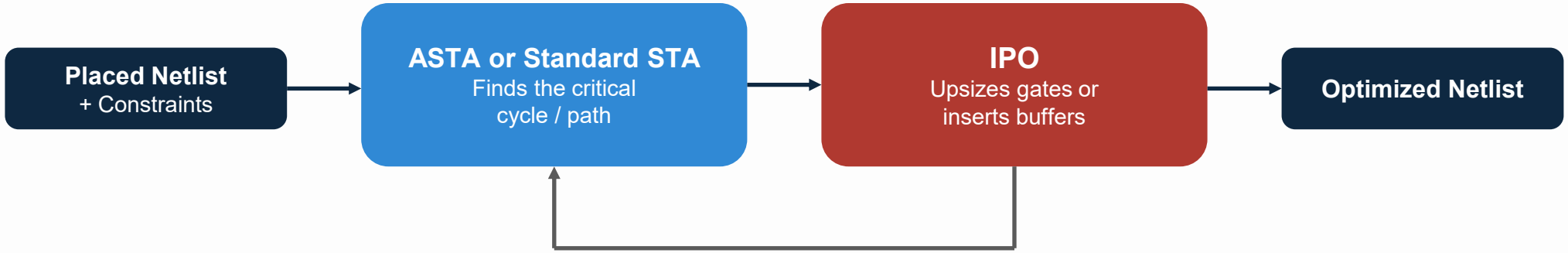
Acyclic graph — sync

✓ **STA works — but the resizer's answer isn't always optimal**

→ AIPO's Cout/Cin-driven upsize/buffer engine optimizes both cyclic and acyclic critical paths

Our Approach: (A)IPO

One optimization engine, two timing back-ends

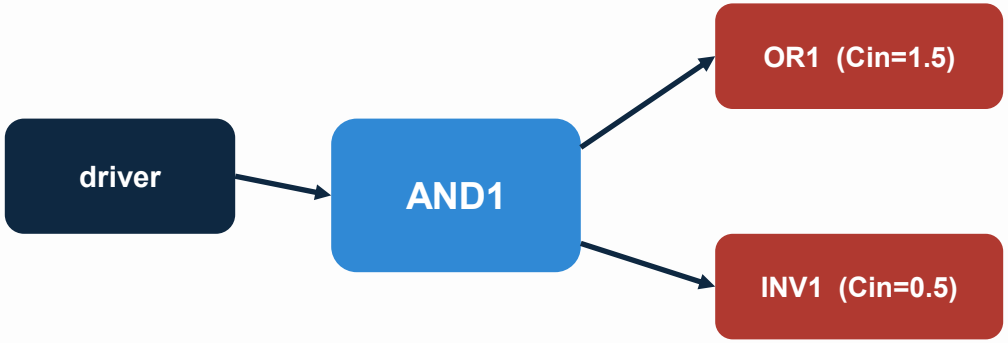


Iterate until the delay target is met (or no more improving moves exist)

Asynchronous mode: our custom ASTA engine identifies the critical cycle
Synchronous mode: a standard STA engine identifies the critical path — the exact same IPO core does the optimizing

Key Idea: the Cout / Cin Ratio

- Computed for every cell on the critical cycle (async) or critical path (sync)
- Cout: total capacitance the cell drives (fan-out gates + wire)
- Cin: the cell's own input drive capability
- **A large Cout / Cin means the cell is overloaded relative to its drive strength**
- **→ a strong candidate for optimization**



Cout = 0.5 + 1.5 + 0.5 (wire) = 2.5 **Cin = 0.30**

Cout / Cin = 2.5 / 0.30 = 8.3 → overloaded

Sync designs — also weight by how many failing paths the cell sits on:
Criticality = (Cout / Cin) × (1 + log(occurrences))

log damps the count — a balanced cell on 500 paths shouldn't outrank a truly overloaded one

Fix 1: Upsize the gate

Fix 2: Insert a buffer

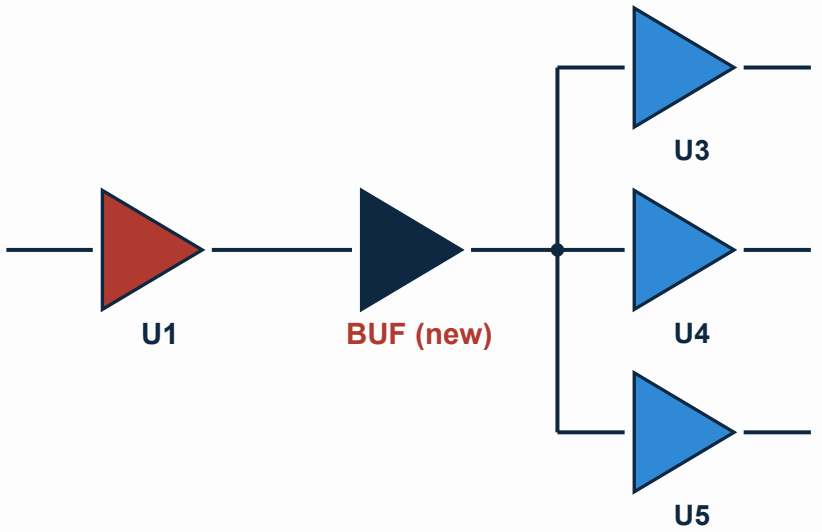
The Two Optimization Moves

- **Move 1: Resizing**
  - Look up functionally-equivalent cells of different drive strengths in the library
  - Score each candidate size locally, and keep the best-scoring one — not necessarily the biggest

Candidate (AND2)	Local Score
AND2_1	1.7
AND2_2	1.5
AND2_4 ← chosen	**1.3**
AND2_8	1.6
AND2_16	2.1

- **Move 2: Buffer Insertion**
  - Used when overload can't be fixed by resizing alone
  - Buffer count: bounded by a max-capacitance cap (MAX_CAP) per cluster
  - Grouping: k-means on cell location keeps each buffer's fanout physically close
  - Worst-slack cluster is left unbuffered, to avoid extra delay where it hurts most

Cout/Cin(U1) = 10.0 → too high to fix by resizing alone



Optimizing Under Power Constraints

Each candidate cell size is judged on both timing and power — blended by a designer-set weight

Step 1: two lookup tables from the library

Timing (delay)

fast → low value
slow → high value
normalized to [0, 1]

Power

low draw → low value
high draw → high value
normalized to [0, 1]

For every candidate size, the library gives us its delay and its power draw. Both are normalized to a 0-to-1 scale so we can mix them.

Step 2: blend with a designer-set weight

Example weights (chosen by designer):

score = (1 − timing) × w_timing + (1 − power) × w_power
highest score wins

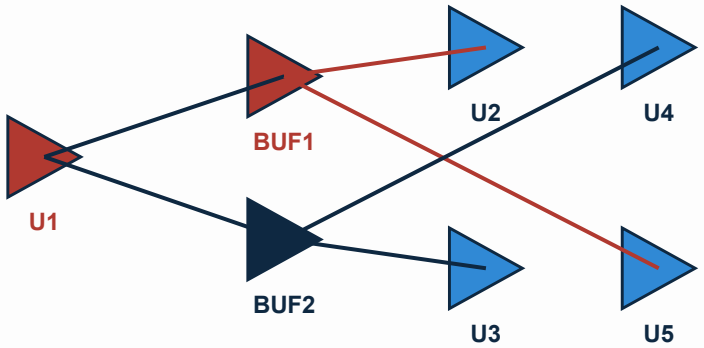
Same weighted score picks gate sizes AND buffer sizes

Result: shifting weight from 100% timing → 60/40 timing/power halves both area (4.4→2.2%) and power (7.5→3.9%) overhead

Buffer Clustering, Step by Step

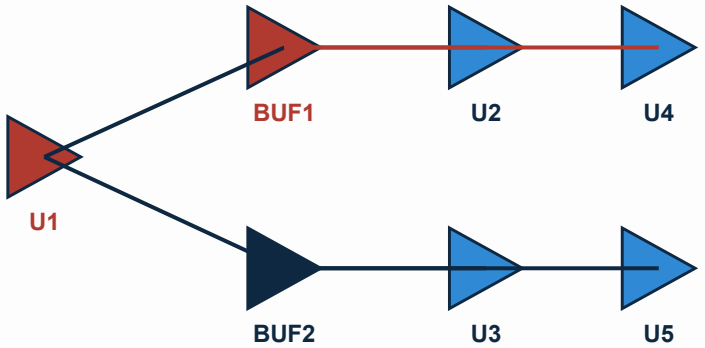
Same driver, same 4 successors — three clustering strategies

1. Capacitance cap only



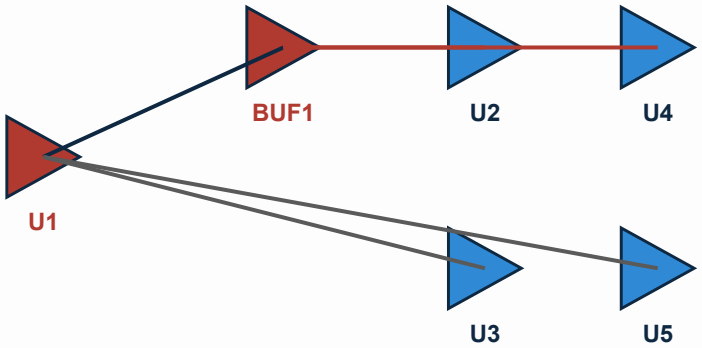
Bad: U2+U5 and U3+U4 are far apart — long, crossing wires

2. + k-means on location



Good: U2+U4 and U3+U5 are each physically close

3. + worst-slack rule

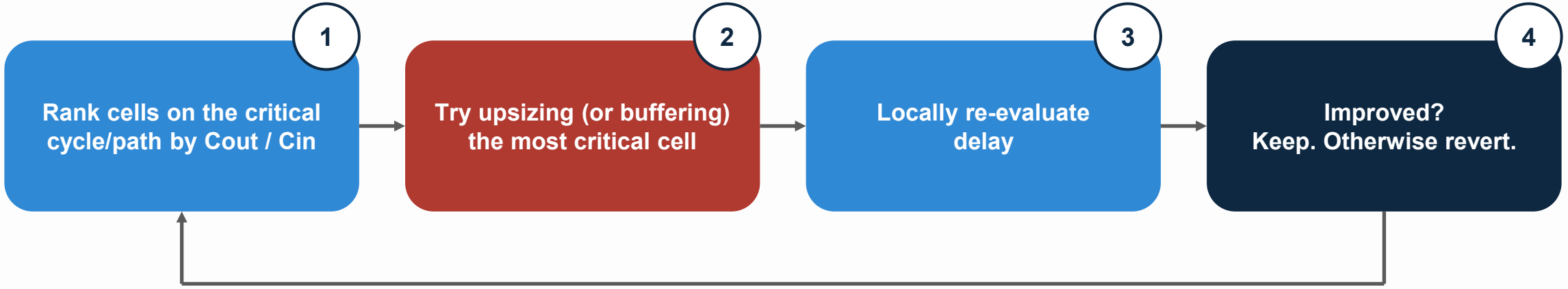


Best: worst-slack pair (U3,U5) stays on the driver — no 2nd buffer

Result: same 4 successors, but the worst-slack rule cuts the buffer count from 2 to 1 — fewer cells, no added delay on the critical path

The IPO Algorithm

An iterative local search over the critical cycle or path



Legalise placement → repeat until the delay target is met or no move improves further

Step 3 swaps in ASTA or standard STA depending on the design — everything else stays identical
Bounded by two user knobs: Max Utilization (placement density) and Max Iterations, so runtime stays predictable

Experimental Setup

- **Asynchronous flow**
  - 17 designs: 13 async controllers + 4 bundled-data datapaths
  - Technology: IHP 250/130nm, GlobalFoundries 22nm
  - Baseline: pre-optimization placed netlist (no other tool can analyze these cycles)
- **Synchronous flow**
  - 9 standard RTL benchmarks: aes, des3, ac97, gcd, tv80, ibex_core, etc.
  - Technology: Skywater 130nm, ASU 7nm
  - Baseline: OpenROAD's own built-in resizer, on the same placed netlist

17 designs 7 – 11.5k cells	**3** tech nodes	**≤80s** max runtime	**9** designs 250 – 75k cells	**2** tech nodes	**1** tool baseline

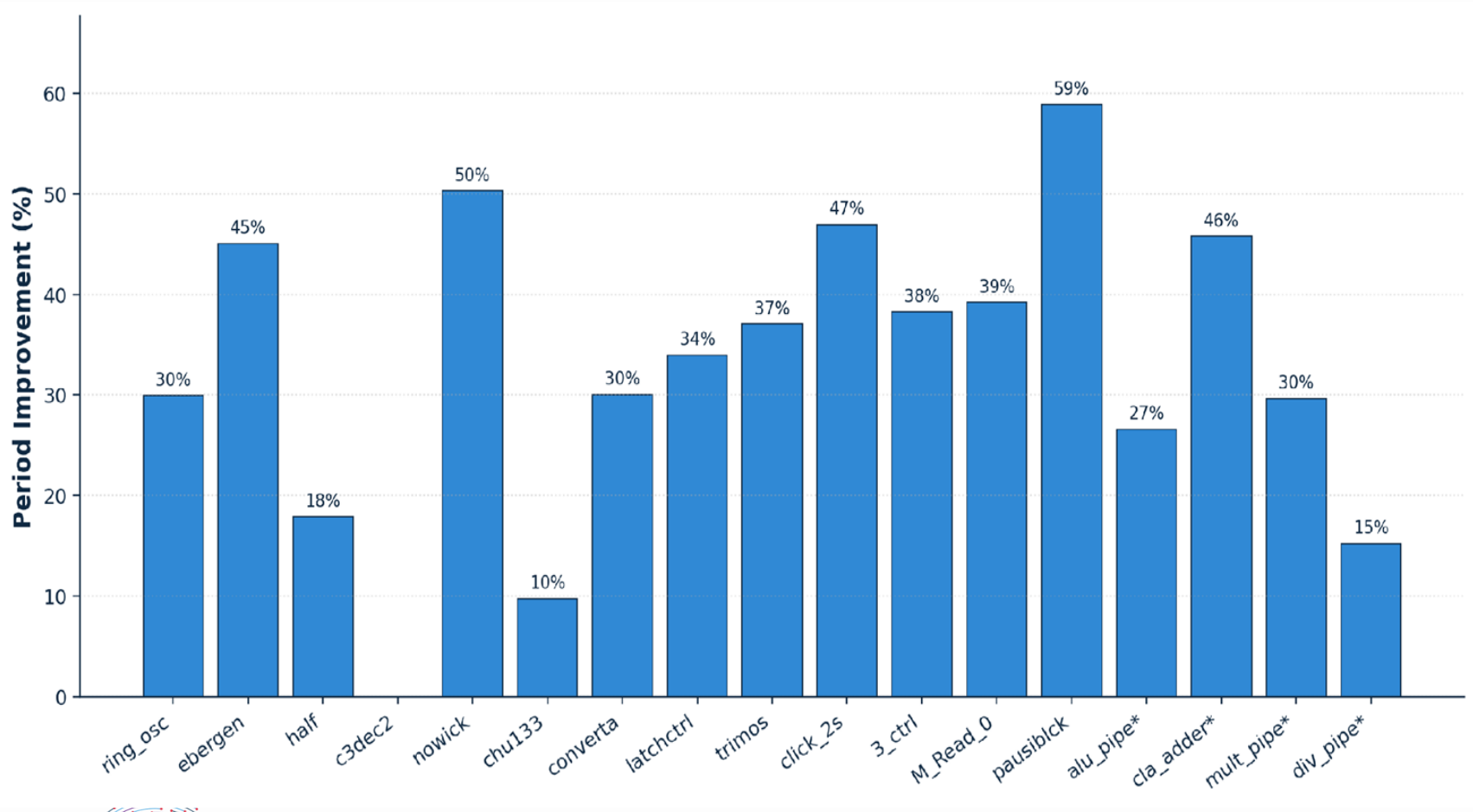
Common tooling: Cadence Innovus for place & route, Synopsys PrimeTime for independent sign-off

For cyclic paths, sign-off uses an iterative slew-override method so PrimeTime can still validate cycles it cannot natively analyze

Runtime: 10-core / 20-thread workstation, 128GB RAM — avg. ~20s and ~40 iterations per design (worst case <80s, <300 iterations)

Results: Asynchronous Designs

Period improvement on 17 cyclic designs — mapped in IHP 250/130nm



IHP
+32.6%
period improvement
5.5% area overhead

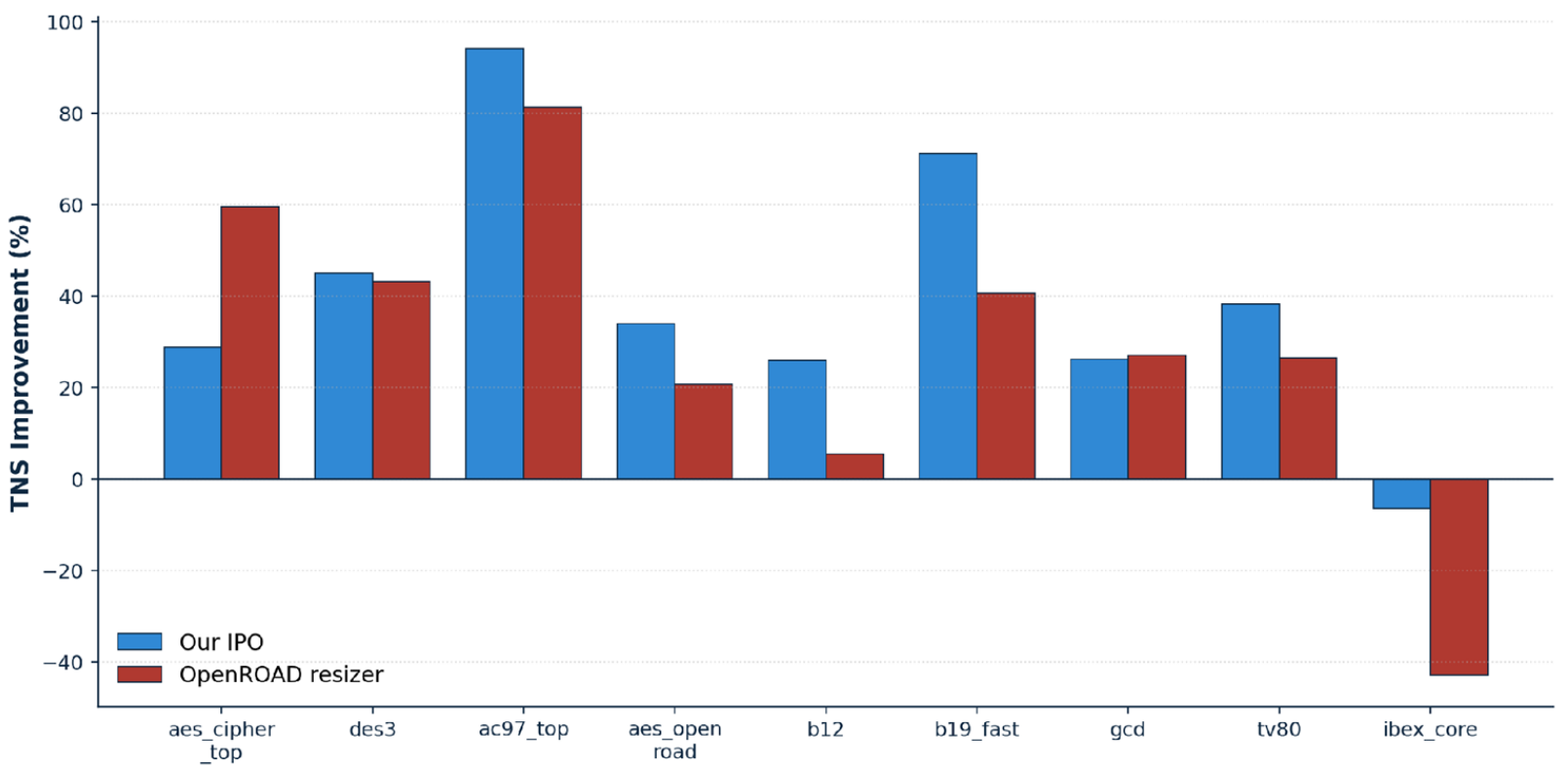
GlobalFoundries 22nm
+29.7%
period improvement
39.6% area overhead*

**small absolute area → large % on tiny designs*

Runtime
~20 s avg, <80 s worst case

Results: Synchronous Designs

TNS improvement vs. OpenROAD's built-in resizer — SKYWATER 130nm



On average, across all 9 designs

Timing (TNS)
Our IPO +39.7%
OpenROAD +29.0%
we're clearly faster

Power
Our IPO +7.5%
OpenROAD +6.1%

Runtime (avg, ASAP7 7nm)
Ours ~400 s vs OpenROAD ~4,260 s

ibex_core was already near-balanced at synthesis — no headroom for either tool

Runtime: ~10× faster than OpenROAD on ASAP7 7nm (65 targeted moves vs ~2,200)

Conclusions & Future Work

- **Conclusions**
  - First post-placement in-place optimization flow for asynchronous, cyclic timing paths
  - The same upsize / buffer engine generalizes cleanly to standard synchronous critical paths
  - Validated against industrial tools: Synopsys PrimeTime, Cadence Innovus, and OpenROAD
  - ~30% average delay improvement in both flows, with area overhead scaling proportionally
- **Future Work**
  - More optimization moves: fanout decomposition, pin reassignment, gate cloning
  - Broader asynchronous timing constraints: isochronic forks, relative timing paths
  - Alternative optimization objectives: power, area
  - Extend STA analysis accuracy: AOCV, POCV, timing models

26
designs validated
(17 async + 9 sync)

~30%
avg. delay
improvement

3
industrial tools
cross-checked

TWINRELECT
TWINNING FOR EXCELLENCE IN RELIABLE ELECTRONICS

Thank you!

Contact:

Dimitrios Tsalapatas

dtsalapatas@uth.gr

ditsalapatas@gmail.com

caslab.e-ce.uth.gr