Deterministic Scheduling

Florian Pollitt

universität freiburg

WG2: Workshop on Automated Reasoning and Proof Logging

September 11.-13., 2025, Orsay, France

supported by DFG Intel

The Problem — Scheduling in SAT solvers

```
CDCL with inprocessing (CNF formula F)
1   (res, F) = preprocess (F, α)     // where α is some estimated effort limit
2   while res = UNKOWN
3       propagate all newly assigned literals
4       if conflicting then res = analyze conflict continue
            // either learn and propagate or return UNSAT
5       if satisfied then res = SAT, continue
6       if restarting then restart, continue
7       if unlearning then unlearn, continue
8       if switching then switch mode, continue
9       if inprocessing then (res, F) = inprocess (F, δ), continue
            // where δ is some effort limit relative to the time spent in propagation
10      decide next unassigned variable
11  return res
```

Stable and unstable mode switching [SAT'15]

- portfolio motivated by differences of SAT/UNSAT
- switching between different heuristics
- decision heuristics
- restart frequency
- more aggressive in unstable mode
- generally slower conflict frequency
- high variance with conflict based schedule
- time based schedule instead

Reproducibility vs. Precision

- Reproducibility for debugging and robustness
- Deterministic counts instead of hardware or operating based solutions

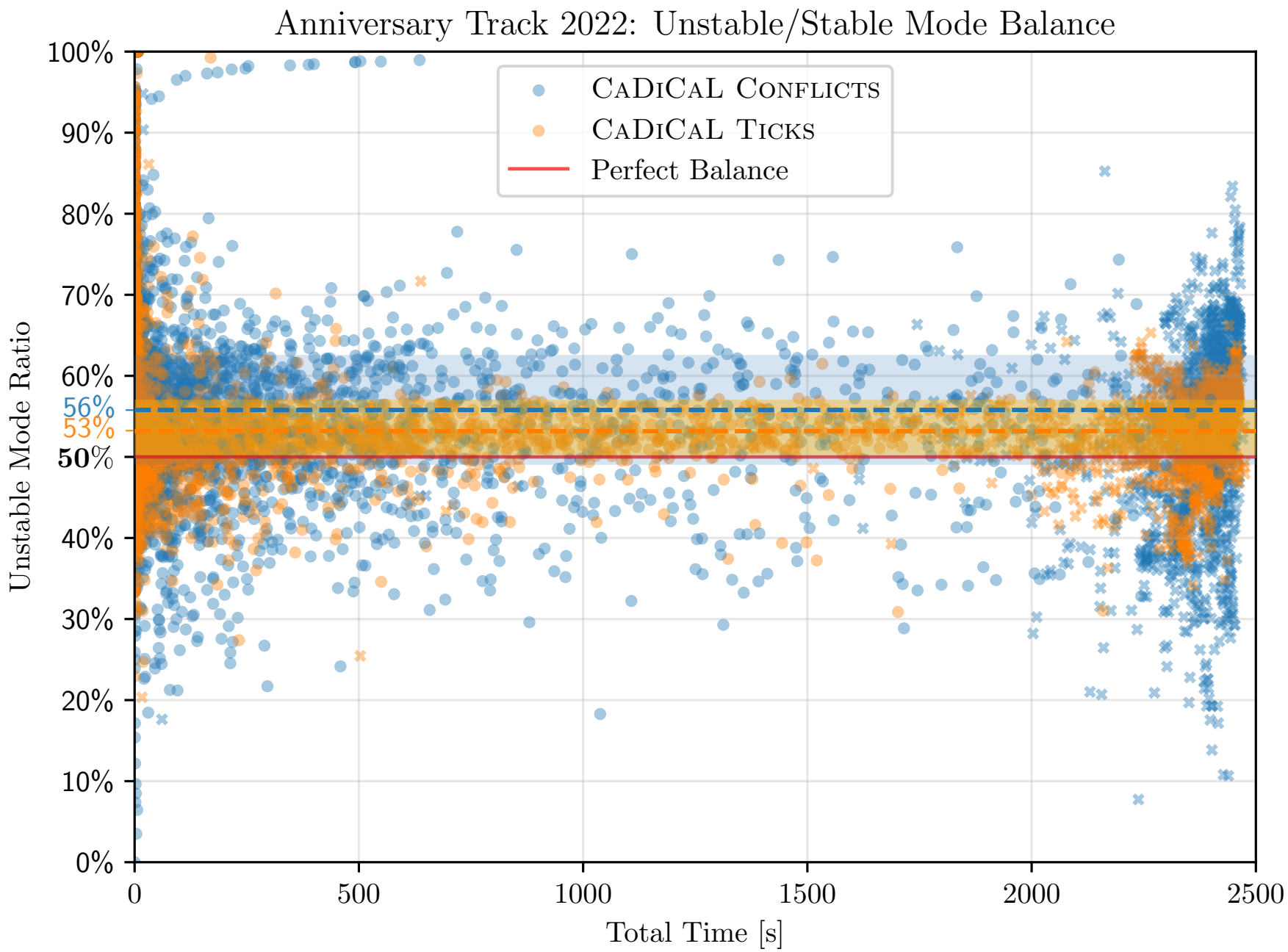
The Approach — Ticks

- widely used metrics are conflicts and propagations
- count cache line accesses in hotspots
- similar to Knuth's *mems*
- (hopefully) better abstraction (takes cache lines into account)
- propagation and clause learning are biggest hotspots

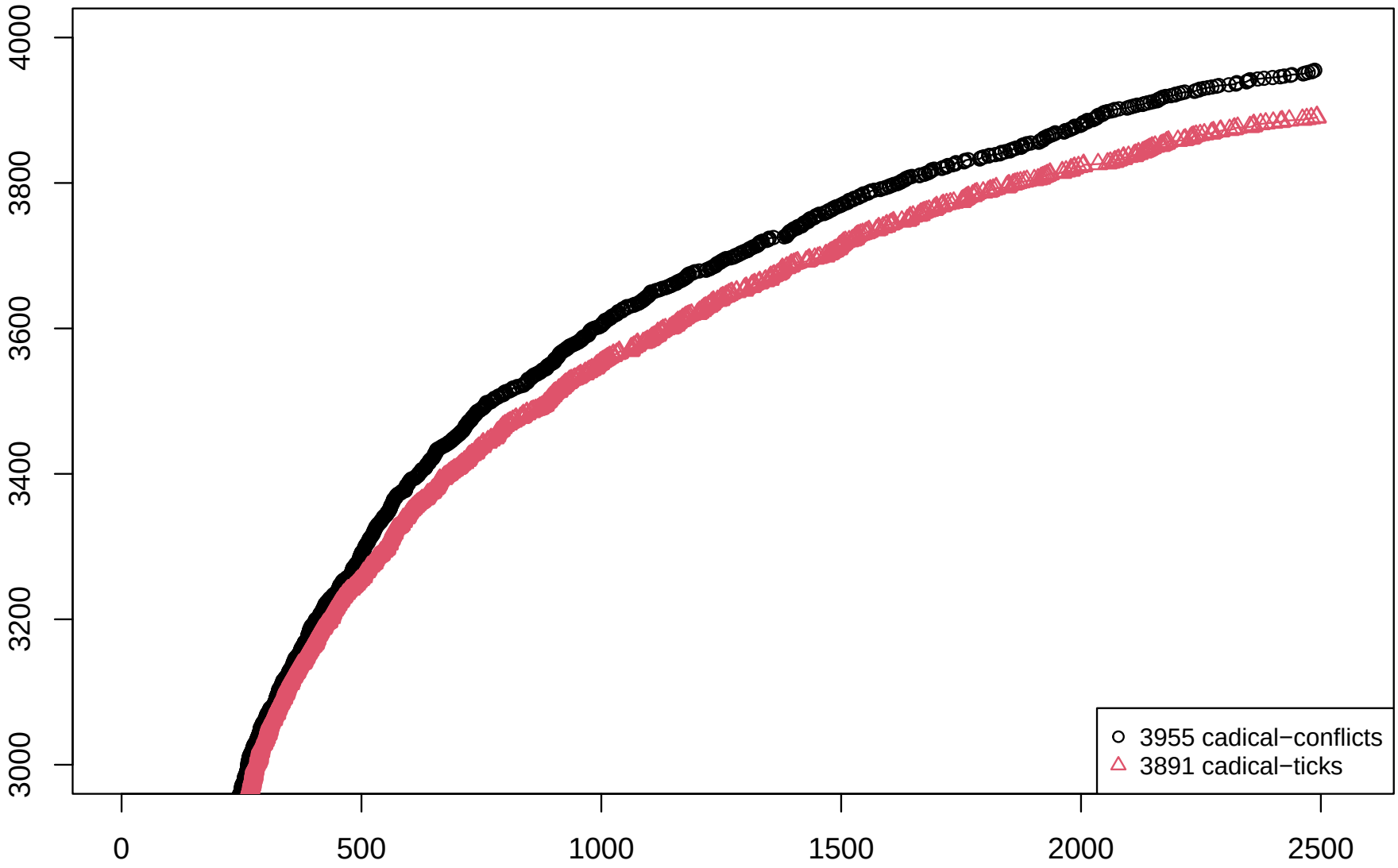
Pseudocode — Propagation with *ticks*

```
   propagate (newly assigned literal ℓ)    // assuming current assignment is cached
 1     using watchlist W of ¬ℓ
 2     let ticks = 1 + cachelines (W)    // assuming 128 byte cachelines
 3     for all watches w ∈ W
 4         if blocking literal of w is satisfied continue
 5         dereference w's clause C, ticks += 1    // main hotspot
 6         other watched literal ℓ_other ∈ C[0 : 1]
 7         find non-falsified replacement literal ℓ′ ∈ C[2 : n]
 8         if valid replacement ℓ′ exists
 9             move watch from ℓ to ℓ′, ticks += 1    // "random" watchlist
10         else if ℓ_other unassigned    // no replacement, clause is propagating
11             assign ℓ_other, ticks += 1    // update reason
12         else break    // clause is conflicting
13     get current mode ∈ {stable, unstable}
14     increment global ticks[mode] by ticks
```

Stable vs. unstable



Is it good though?



Inprocessing

- do not get “stuck” at inprocessing
- correlate inprocessing time with soving time
- fixed usefullness fraction
- bounded variable elimination
- vivification
- hyper binary resolution
- . . .

Integrating ticks — Profiling

- add ticks to presumed hotspots of algorithms
- check precision by running benchmarks with profiling
- target relative time vs. actual times
- refine by looking at code

Conclusion

- more precise then conflicts/resolutions
- disregards benchmark characteristics
- manual effort (bug prone)
- lots of profiling (CPU time)
- “good enough” solution
- does not work for up-front effort (preprocessing)

Bibliography

[2011] *Donald Knuth*.
“The Art of Computer Programming, Volume 4A”

[SAT’15] *Chanseok Oh*.
“Between SAT and UNSAT: the fundamental difference in CDCL SAT”