WG5: Applications & Performance Evaluation

Pascal Felber
pascal.felber@unine.ch

Goals of STM

- Simplify concurrent computing...
  (like sequential programming)
  ...while providing good performance...
  (like fine-grained locking or custom concurrent algorithms)
  ...and “usable” semantics...
  (e.g., progress)
  ...for a wide range of applications
- Are these goals compatible?

Simplicity?
(vs. semantics)

Simplicity?
(vs. semantics)

Strong atomicity

Open nesting

Blocking

Obstruction-free

Linearizable

Opaque

Weak atomicity

Snapshot isolation

Closed nesting

Serializable

Disjoint-access-parallel

“Almost” wait-free

Publication

Boosting

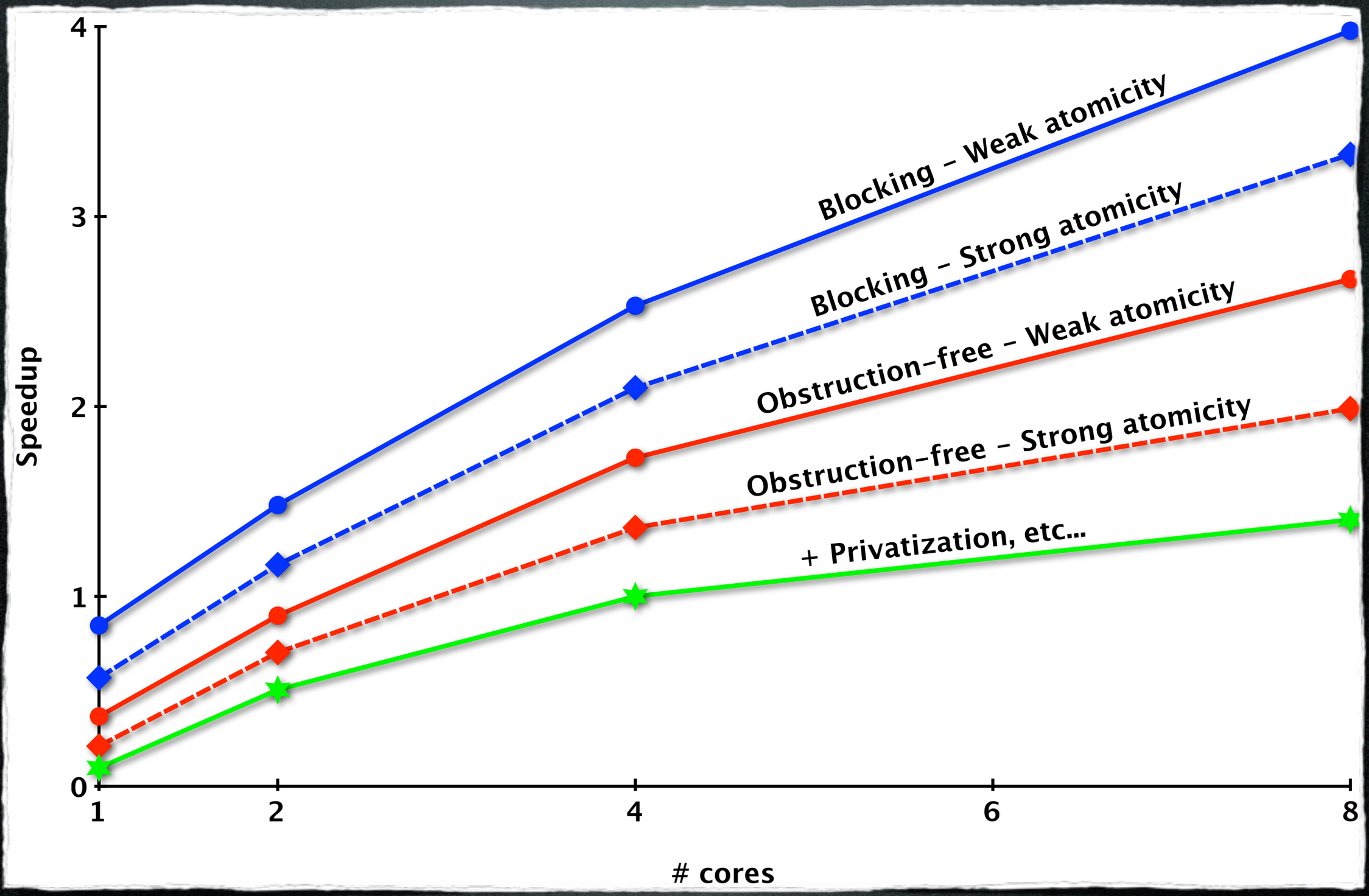
Privatization

Performance?
(vs. semantics)

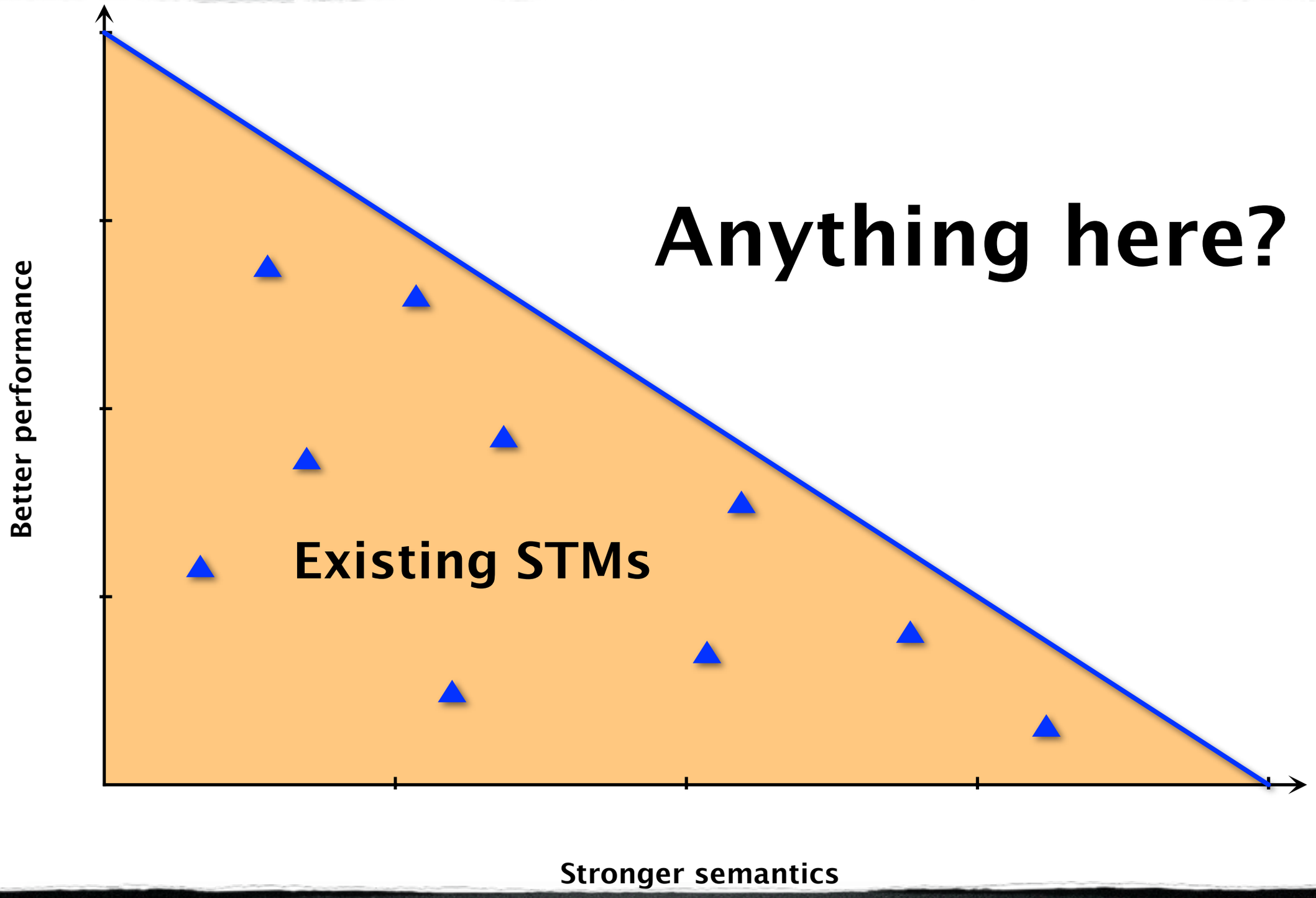
- Performance: what metrics?
  - Throughput vs. scalability vs. #aborts? Progress? Fairness?
- Also depends on semantics
  - Weaker semantics make TM faster... (less guarantees to provide)

    ...but make programming harder (make sure application remains correct)

Performance?
(vs. semantics)



Performance?
(vs. semantics)



Applications?

- TM is not good for all applications
- There should be some conflicts...
  (otherwise no synchronization necessary)

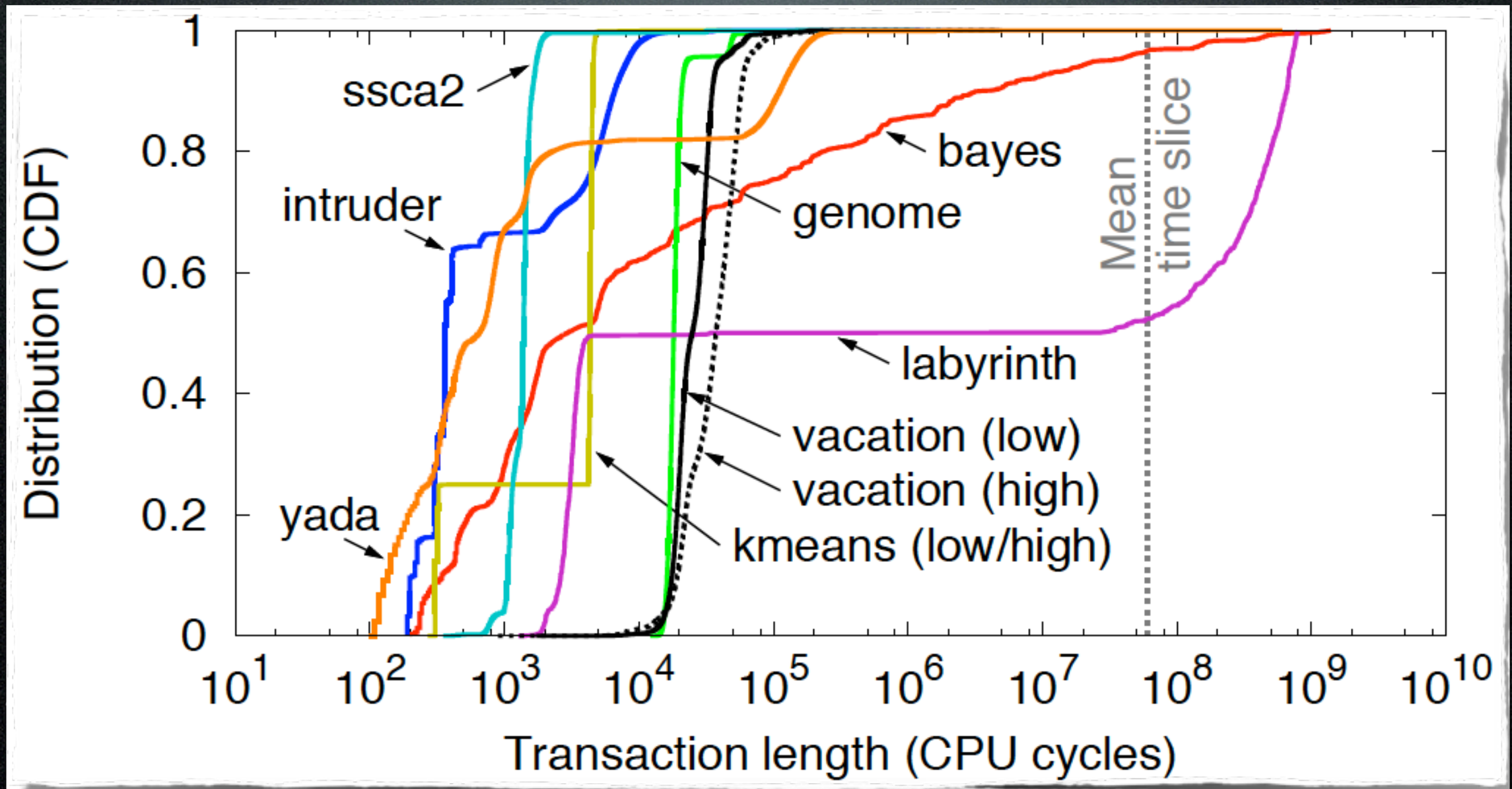
...but not too many...
(otherwise pessimistic CC is better)

...with not-too-long transactions...
(to keep the cost of aborts reasonable)

...involving data not known statically
(otherwise no simpler than locks)

Workload properties

Transaction length, number of atomic blocks and call frequency, read-only vs. update, ...



1: Snapshot isolation

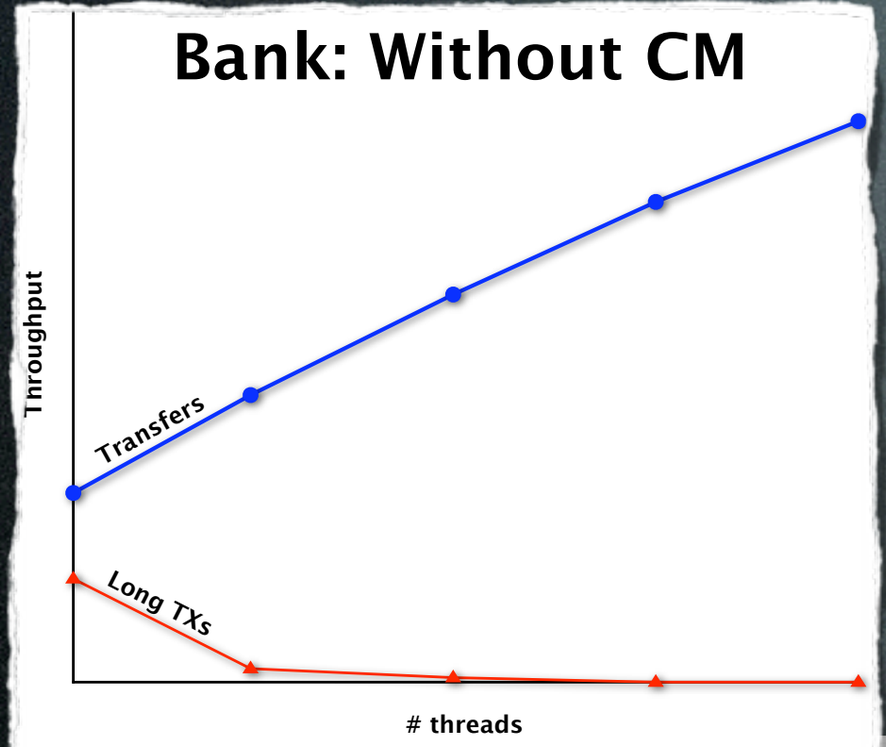
- Widely used in DBs
  - Read from initial snapshot, commit if no write-write conflict
  - In TM, performance gain negligible, higher programming complexity
    - Must add writes to force conflicts
- **Bottom line: SI not (very) useful for TMs, bad for programmer**

2: Serializability

- More concurrency, sufficiently strong for the programmer
- Must keep track of conflict graph
  - Runtime overhead, little benefits (higher C-A ratio, lower throughput!)
  - Useful only if aborts are costly (e.g., ms+ transactions)
    - Right workload for TM?
- **Bottom line: not very useful for TMs**

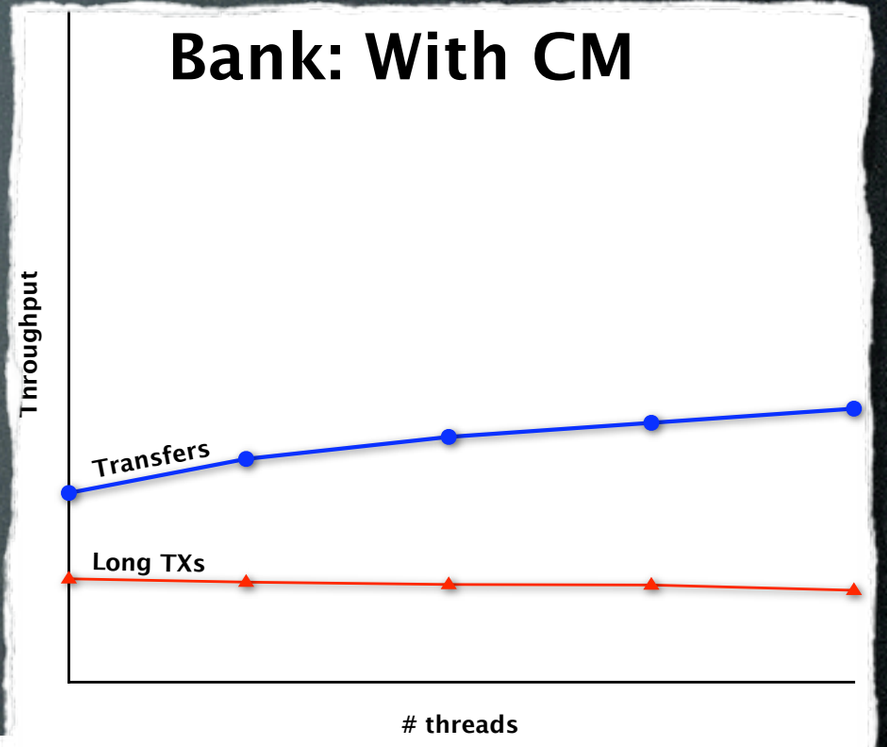
3. CM

- Many CM proposed in the literature
  - Kill self/other, older, shorter, nearer to completion, ..., or wait
  - Differ in terms of complexity and (progress) guarantees
- Performance of CM depends on the types of conflicts in the workload

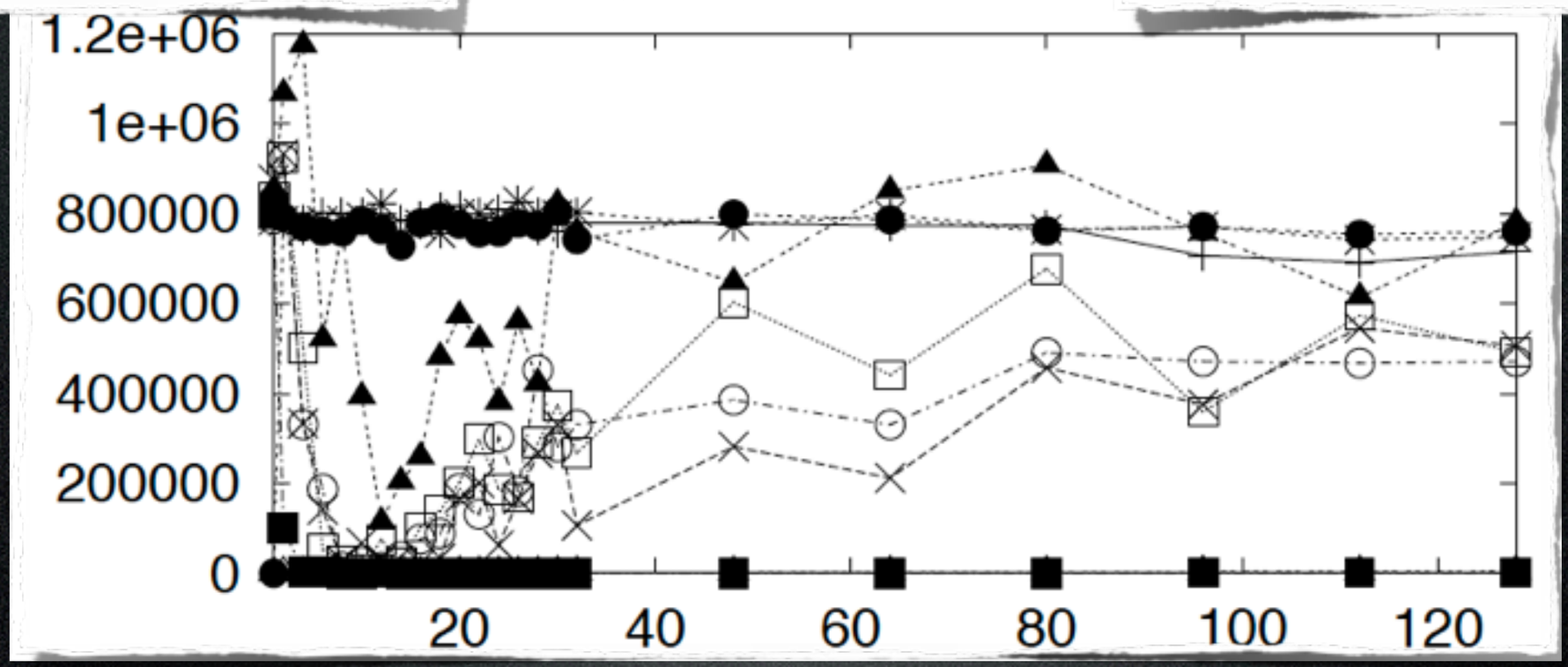


3. CM

Fairness between different **types** of transactions often degrades "raw throughput" (right metrics?)



Any clear winner?
[Scherer,Scott]



- **Bottom line: unless fairness required, use simple all-around CM**

4. Early release & elastic

- Early release tells TM that memory location will not be accessed anymore
  - Not trivial to use (2PL)
- Elastic transactions can be cut at runtime into sub-transactions
  - Must tag elastic transactions
- **Bottom line: better performance, less conflicts, but harder to use**

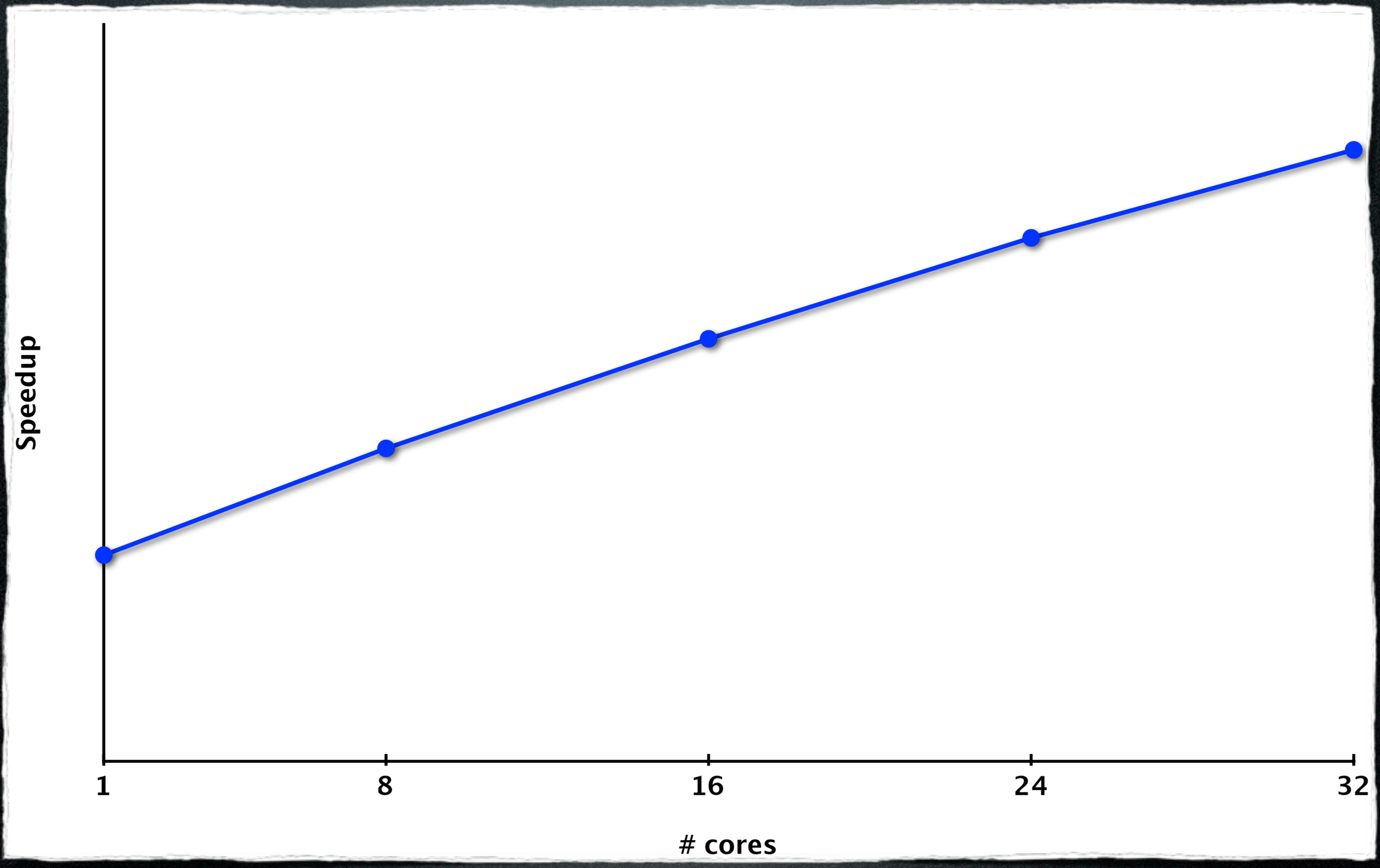
Which semantics?

- Keep it simple (for the programmer)
  - E.g., snapshot isolation, causal serializability **hard** to reason about
  - E.g., SGLA: **familiar** to developer, **simple** operational semantics
- Weak is fine if **well specified**, **easy to use**, and noticeable **performance gain**
  - What is dis-/allowed by a specific model

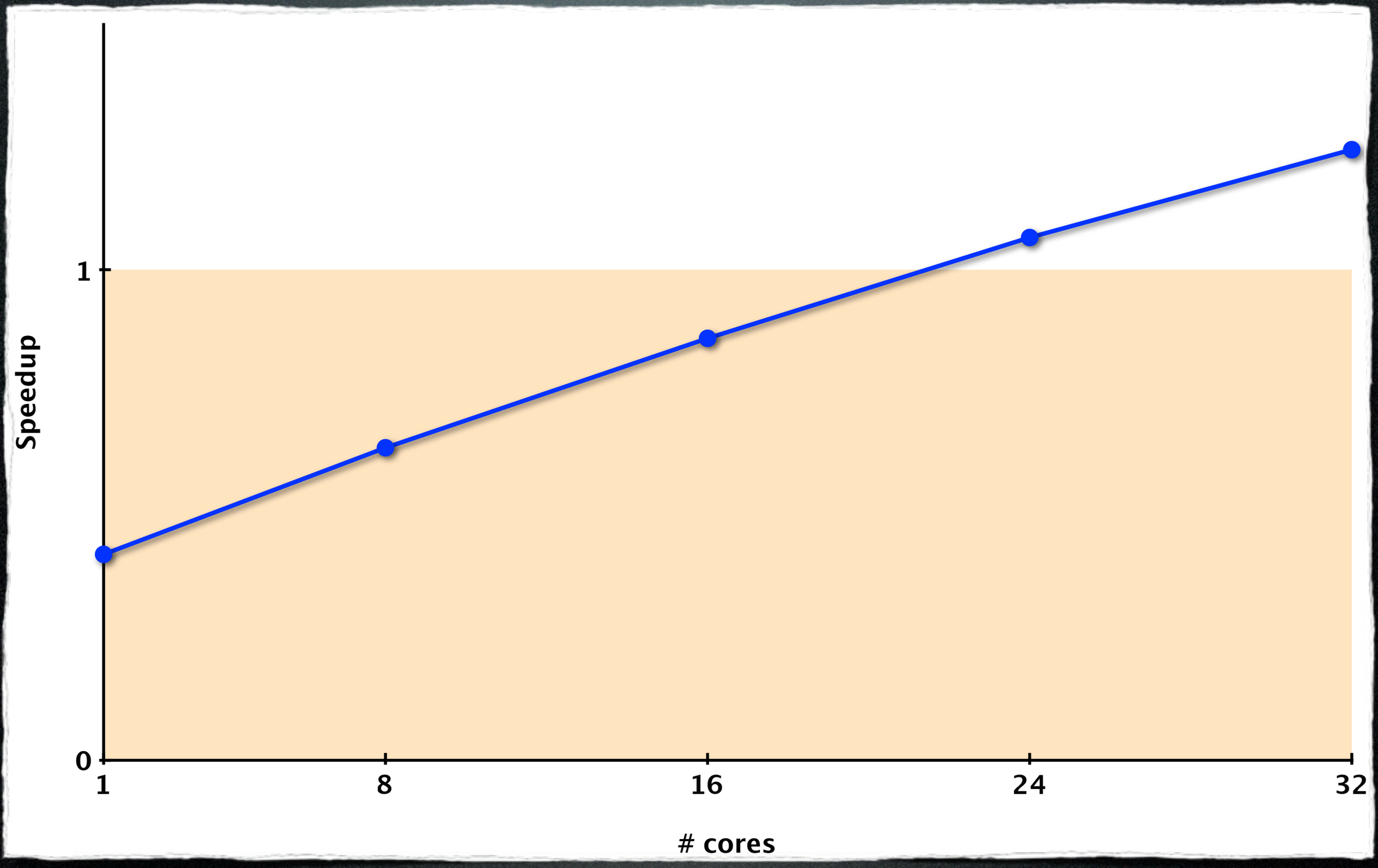
What performance?

- Even more than simplicity of use, **good performance** is necessary for wide adoption of STMs
  - Need HTM or HyTM?
- Performance measured in terms of
  - **Scalability** (exploit many cores)
  - **Speedup** over sequential

Scalable ≠ fast



Scalable ≠ fast



Wrapping up

- Balance between simple semantics and efficient implementation
- Adoption: standardization, (multiple) language support, HW support?
- There is no one-size-fits-all TM
  - Effectiveness depends on **workload**
  - Often: **simple+scalable** but **slow**
- Still looking for good TM applications!